

Computer Composition Principle Online Experiment Course

Teaching and Practice of FPGA Remote Experiment Platform

计算机组成原理在线实验教程

FPGA远程实验平台教学与实践

◎ 柴志雷 李佩琦 吴子刚 阳文敏 编著

Chai Zhilei Li Peiqi Wu Zigang Yang Wenmin



- “线上线下结合、课内课外一体化”的实验环境
- 基于RISC-V及开源项目进行实验
- 采用Python程序与所设计的硬件模块进行交互

清华大学出版社

面向“工程教育认证”计算机系列课程规划教材

计算机组成原理在线实验教程
——FPGA 远程实验平台教学与实践

柴志雷　李佩琦　吴子刚　阳文敏　编著

清华大学出版社
北　京

内 容 简 介

本书以线上线下结合的方式，使用FPGA平台完成数字电路及计算机组成原理实验，并用Python编程的方式与自己设计的硬件系统交互，来进行系统验证与调试。全书包含三大部分内容：首先是实验所用的软硬件工具和平台；其次是实验设计方法；最后介绍实验内容安排的建议。书中主要讲述了PYNQ平台与基于Python的软硬件交互、Vivado开发工具、硬件描述语言简介、RISC-V开源项目及组成原理实验内容建议等。

本书可用作高等院校计算机类、电子信息类专业的数字电路与组成原理相关课程的实验教材，也可作为FPGA及嵌入式系统软硬件学习的参考用书。

图书在版编目(CIP)数据

计算机组成原理在线实验教程：FPGA远程实验平台教学与实践/柴志雷等编著. —北京：清华大学出版社，2020.1
面向"工程教育认证"计算机系列课程规划教材
ISBN 978-7-302-53779-3

Ⅰ. ①计…　Ⅱ. ①柴…　Ⅲ. ①计算机组成原理—实验—高等学校—教材　Ⅳ. ①TP301-33

中国版本图书馆CIP数据核字(2019)第200064号

责任编辑：刘　星
封面设计：常雪影
责任校对：梁　毅
责任印制：李红英

出版发行：清华大学出版社
　　网　　址：http://www.tup.com.cn，http://www.wqbook.com
　　地　　址：北京清华大学学研大厦A座　　**邮　　编**：100084
　　社 总 机：010-62770175　　**邮　　购**：010-62786544
　　投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
　　质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
　　课件下载：http://www.tup.com.cn，010-83470236
印 装 者：清华大学印刷厂
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　张**：11　　**字　　数**：265千字
版　　次：2020年1月第1版　　**印　　次**：2020年1月第1次印刷
定　　价：39.00元

产品编号：068985-01

前言

PREFACE

随着人类社会迈向万物互联的物联网时代，借由人工智能技术替代手工方式对大数据进行高效实时的处理已成为社会发展的必然选择，万物互联的未来之路一定走向万物智能，而人工智能的构成要素包含数据、算法及计算力。在此背景下，无处不在的人工智能对计算系统的计算能力提出了挑战性的需求，目前如火如荼的边缘计算即是对这一挑战的积极应对。而使得计算力需求更加充满挑战的原因是：长期以来半导体行业遵循的摩尔定律和登纳德缩放定律已难以为继，这意味着芯片计算能力的提升已经无法在保持架构基本不变的情况下仅靠工艺的提升来完成，也意味着未来我们将更多地需要依赖架构创新，为应用定制芯片、根据应用需求深度优化软硬件系统。因此高校培养的人才也要比过去更多地掌握硬件设计及软硬件协同优化的知识，这也是近年来教育部高等学校计算机类教学指导委员会一直在大力推进“计算机系统能力”培养的原因。

经过若干年的努力，与系统能力紧密相关的课程，尤其是以“计算机组成原理”为代表的课程的教学及实践已经有了长足的进步。越来越多的学校开始采用 FPGA 作为实验教学的平台，并尝试将“数字逻辑”“计算机组成”“体系结构”“编译原理”甚至“操作系统”等多门课程的实践环节贯通，以此培养学生的软硬件系统能力。以清华大学、北京航空航天大学、浙江大学、华中科技大学、东南大学、同济大学等为代表的一些高校在此方面做了大量的工作并已取得了丰硕的成果。

总体上看，目前围绕计算机组成原理进行的软硬件系统能力培养具有如下特点：

(1) 从实验平台来看，已采用 FPGA 的学校主体上仍是以线下的 FPGA 实验设备为载体进行实验，从时间和空间上限制了学生像软件类的实验一样随时随地进行。学生实验的达成度严重受限于实验课时。

(2) 从实验的调试和交互方式来看，目前的实验主要使用 FPGA 实验装置上的拨码开关、LED、串口等物理 I/O 进行，由于数量有限，极大地影响了调试的便利性。

(3) 从实验内容来看，鉴于《计算机组成与设计：软硬件接口》一书的巨大影响力，多数学校选择的实验内容是兼容 MIPS 指令集的处理器设计。虽然 MIPS 指令集非常经典，用于教学时理论及实践素材都十分丰富，但难掩其产业界影响偏小、产学融合深度和广度有限的缺憾。因此常见的组合是，在讲授组成与体系结构时采用 MIPS 架构，但在讲授后续嵌入式系统时却几乎都切换成了 ARM 体系。

因此，为了结合人工智能时代计算系统的发展趋势进一步提升学生的系统能力，在前期教指委推动的系统能力培养工作基础上，计算机组成原理的实验教学还可以在如下方面进一步提升：

(1) 实验平台方面：探索“线上线下结合、课内课外一体化”的实验环境。课内采用线下方式，让学生对真实硬件有更直观的认识。课外采用在线实验，使学生可以随时随地通过远程访问实验设备进行实验。线上线下的使用方式保持一致，从而为学生提供可与软件实验相媲美的实验条件。尽管目前越来越多的高校认识到相比传统的插线式实验平台，采用FPGA 进行组成原理实验教学将是未来的发展趋势。但现状是为数不少的学校仍然采用的是传统的插线式平台，在切换到 FPGA 方式时缺少便捷的“试水”途径。因此，线上方式及相应的参考内容可为系统能力培养较为薄弱的地区提供一种易于推进的模式。

(2) 实验的调试和交互方式：本书采用 Python 程序与所设计的硬件模块进行交互。鉴于 Python 语言已成为人工智能时代最为流行的编程语言，使用 Python 可以突破实验平台物理 I/O 的局限，在提升交互便利性的同时可锻炼学生的软硬件协同设计能力，为学生今后从事人工智能计算相关的工作奠定良好的系统基础。对于线上方式，更没有必要刻板地模仿线下设备的物理 I/O 交互却又难以达到线下使用的真实感。

(3) 实验内容和设计方法：本书介绍基于 RISC-V 的开源开放生态的处理器设计。鉴于开放指令集 RISC-V 在学术、工业界关注度的急剧增长及众多 RISC-V 指令集的开源处理器项目的快速发展，从目前看其最有希望构建起允许学生探究底层实现细节并促进产学深度融合的生态系统，而这是目前其他指令集系统不具备的优势。在硬件开源开放的背景下，虽然逐条增加指令的增量式处理器设计方法依然重要，但如何基于开源资源快速完成自己的系统也应该开始引起更多的重视。因此，本书建议的实验内容由浅入深，最终介绍如何基于 RISC-V 及开源项目进行实验，以使学生的所学在未来可以更好地和企业界进行对接。当然，本书提供的实验范例也仅供参考，各学校目前所采用的内容也都可以方便地在本书所介绍的平台上进行。

本书介绍如何以线上及线下方式使用 PYNQ 平台进行实验。PYNQ 可以支持 Python，它的核心芯片为 Xilinx 的 SoC 方式的 Zynq FPGA。基于该实验平台，不仅可以满足本书中冯·诺依曼架构的经典处理器的实验教学需求，而且为学生进一步学习非冯·诺依曼架构的硬件加速器奠定好实验平台的基础。在当前人工智能发展对计算能力不断提出更高需求的大背景下，针对特定问题设计专用加速器成为趋势，而这些专用加速器通常都是非冯·诺依曼架构的。因此，学生在学习计算机组成原理课程，掌握经典的冯·诺依曼架构之后，进一步学习突破冯·诺依曼架构限制的专用加速器的知识是培养学生系统能力的未来趋势。

本书包含三大部分内容：首先是实验所用的平台及开发工具；其次是设计方法；最后介绍实验内容安排的建议。具体如下：

第 1 章　PYNQ 开发平台介绍。本章介绍本书实验所用的 PYNQ 平台，除介绍 PYNQ 开发板及开发板所用的主芯片 Zynq-7020 FPGA 芯片之外，还介绍了该款开发板所支持的 PYNQ 框架及其理念。

第 2 章　实验环境的准备。本章介绍在采用 PYNQ 进行组成原理实验时，线下和线上的实验环境分别如何准备，如何通过线上或线下的方式使用 PYNQ 平台。

第 3 章　基于 PYNQ 的组成原理实验流程概览。本章主要介绍本书线上及线下所用的 FPGA 实验平台——PYNQ 上组成原理实验的概要流程，为读者建立一个整体的概念，

方便理解后续的章节内容。

第 4 章　Vivado 开发流程。本章介绍在 PYNQ 平台上进行组成原理实验时，处理器及硬件模块设计所用的工具——Vivado，详细介绍使用 Vivado 进行开发时的流程及一些关键开发知识。

第 5 章　基于 Python 的 I/O 交互。本章主要介绍如何基于 Jupyter Notebooks，通过编写 Python 代码和用户设计的硬件系统进行交互，完成组成原理实验的验证和调试。

第 6 章　硬件描述语言简介。本章主要介绍 Verilog/VHDL 语言的基本设计方法和基本模块示例，更详细的介绍请参考相关书籍。

第 7 章　基于开源 CPU 的组成原理实验。本章介绍基于开源指令集 RISC-V 及开源的 RISC-V 处理器项目进行组成原理实验的理念及设计方法。同时为了扩展读者的知识面，也简单介绍了常规的逐条增加指令的增量式设计方法。

第 8 章　实验内容设计。本章给出一些供参考的组成原理实验内容的设计，高校在开展组成原理实验教学时可以直接选用本章的实验，也可以根据各自学校的实际情况进行内容的重新设计和规划。

本书前言部分及第 1～5、7、8 章由柴志雷编写；第 6 章及第 4、5、7、8 章中的代码部分由李佩琦、吴子刚编写；阳文敏提供了在线平台的支持和实验内容的审核；李康等也参与了部分内容的编写和校对；中北大学的秦品乐教授和于一老师参与了本书的审阅并对内容安排及呈现方式提出了宝贵的修改建议。全书由柴志雷统稿及审阅。在本书内容设计及成稿的过程中，江南大学计算机科学与技术专业的本科生提供了宝贵的反馈意见，唐雨馨、刘昊鑫、李莹莹、李慧琳等同学参与了实验案例的设计。在本书的撰写过程中，得到了江南大学物联网学院及计算机科学与技术系的大力支持。本书的工作还得到了教育部高等学校计算机类专业教学指导委员会“系统能力培养”课程建设试点院校项目、Xilinx—教育部产学合作教学改革项目的支持。在成书的过程中还得到了南京大学袁春风教授、太原理工大学强彦教授、赛灵思(Xilinx)创新研究院陆佳华先生等众多专家的鼓励与支持。在此一并表示感谢。

本书中涉及的一些源代码及资源，请在清华大学出版社官方网站本书页面获取。

最后，受个人能力所限，本书的内容难免会存在不妥之处，敬请读者批评指正，我们将会持续改进。

作　者

2019 年 7 月

目录
CONTENTS

第 1 章 PYNQ 开发平台介绍

CHAPTER 1

1.1 PYNQ 开发板

1.1.1 PYNQ Z1

PYNQ Z1 是由美国迪芝伦公司(Digilent)推出的一款支持 PYNQ 框架(PYNQ 框架将在 1.3 节中介绍)的 FPGA 开发平台,其开发板如图 1-1 所示。

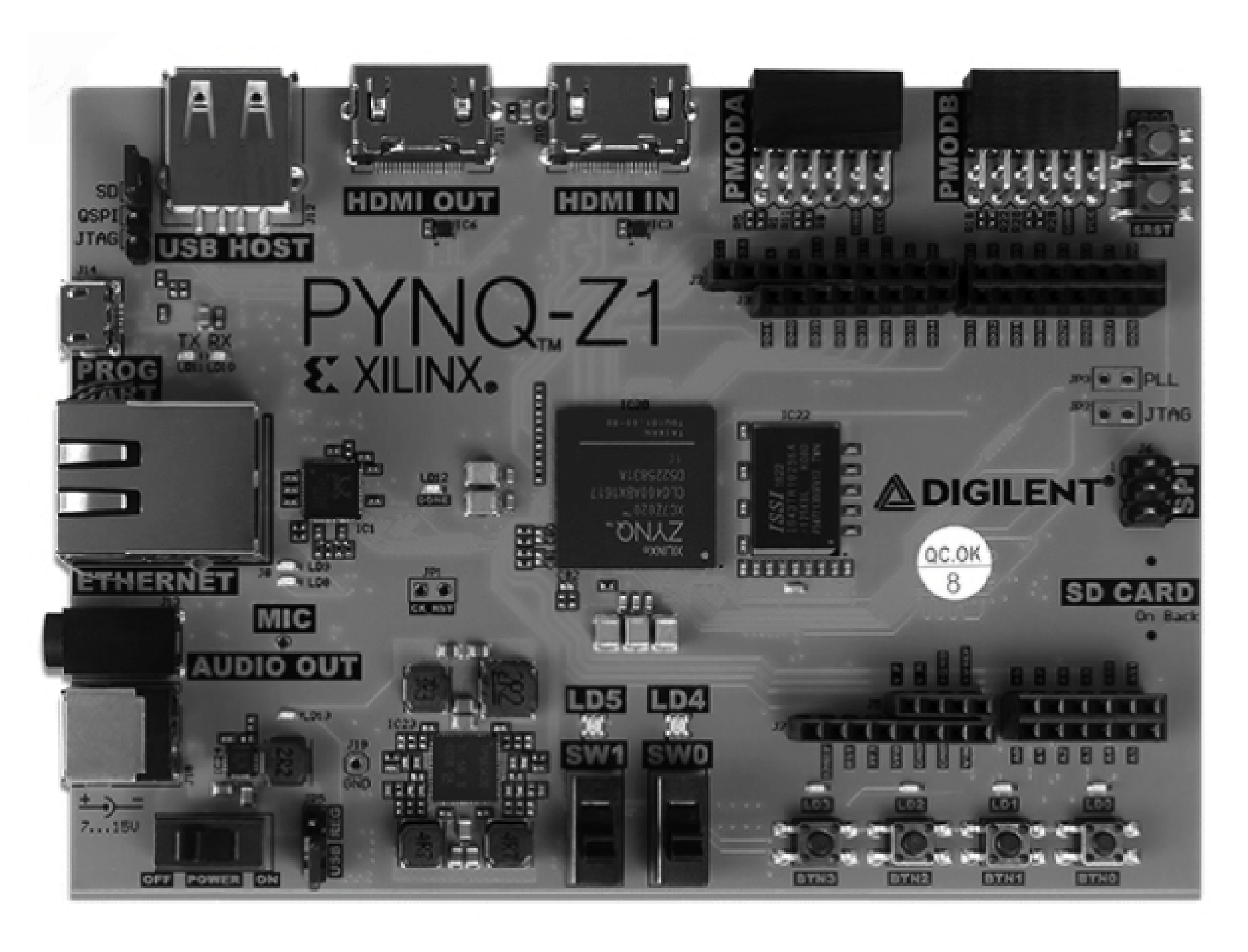


图 1-1　PYNQ Z1 开发板

PYNQ Z1 的具体配置如下：

(1) 核心芯片：Zynq XC7Z020-1CLG400C。

(2) 存储：

- 带有 16 位总线@1050Mbps 的 512MB DDR3。
- 16MB Quad-SPI 闪存,具有出厂编程的全球唯一标识符(兼容 48 位 EUI-48/64™)。
- MicroSD 插槽。

(3) 供电：由 USB 或任何 7～15V 电源供电。

(4) USB 和以太网：

- 千兆以太网 PHY。

• USB-JTAG 编程电路。
• USB-UART 桥。
• USB OTG PHY(仅支持主机)。

(5) 音频和视频:
• 具有脉冲密度调制(PDM)输出的驻极体麦克风。
• 3.5mm 单声道音频输出插孔。
• HDMI 接收端口(输入)。
• HDMI 源端口(输出)。

(6) 开关、按钮和 LED:
• 4 个按钮。
• 2 个滑动开关。
• 4 个 LED。
• 2 个 RGB LED。

(7) 扩展连接器:
• 两个标准 Pmod 端口。
• 16 个 FPGA I/O 接口(与树莓派接口共享 8 个引脚)。

(8) Arduino/chipKIT 屏蔽连接器:
• 49 个 FPGA I/O 口。
• 6 个 XADC 的单端 0~3.3V 模拟输入。
• 4 个 XADC 的差分 0~1.0V 模拟输入。

1.1.2 PYNQ Z2

PYNQ Z2 是台湾厂商 TUL 推出的支持 PYNQ 开源框架的第二代开发平台,其开发板如图 1-2 所示。

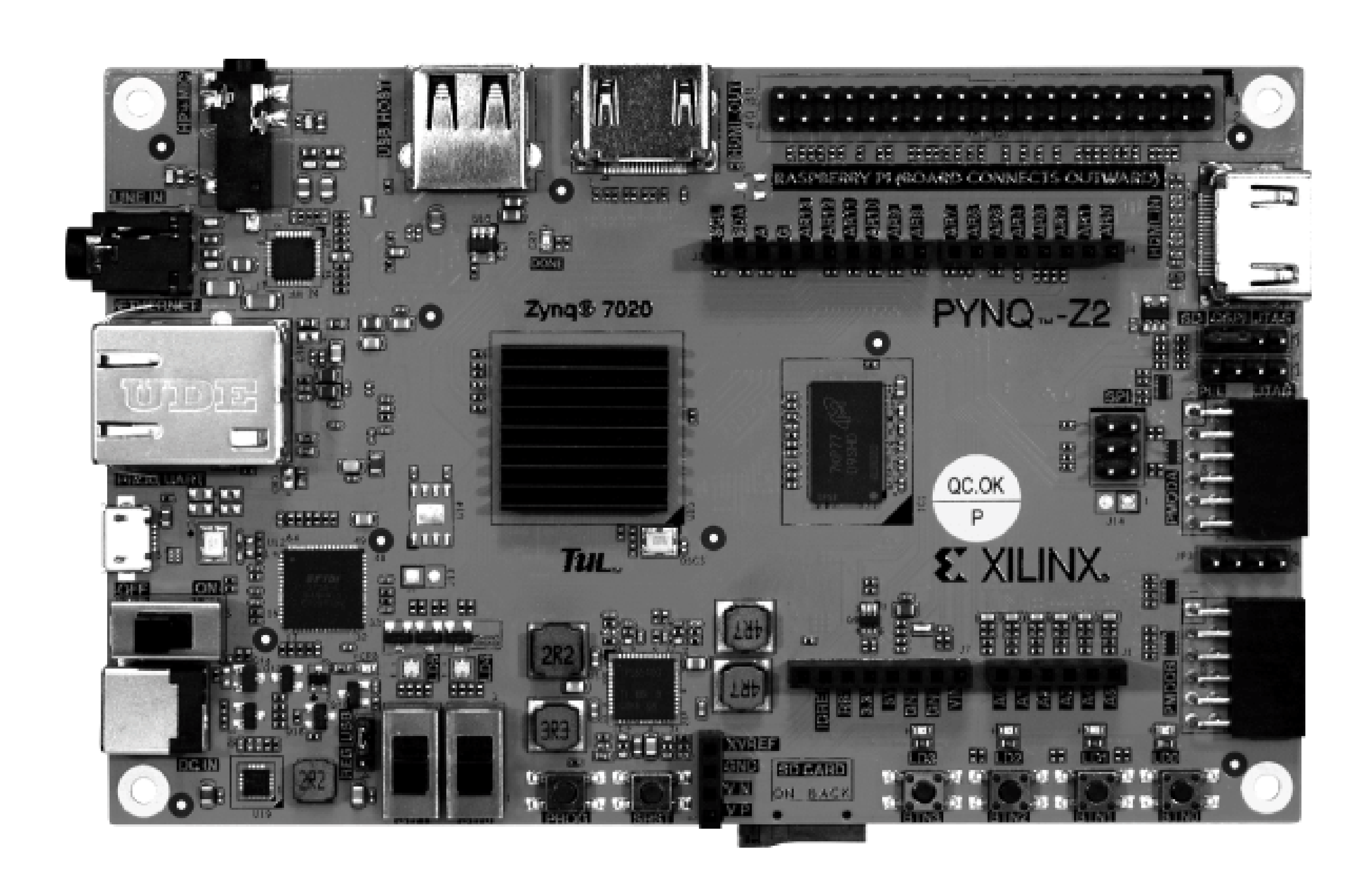


图 1-2 PYNQ Z2 开发板

PYNQ Z2 的具体配置如下:

(1) 核心芯片:Zynq XC7Z020-1CLG400C。

(2) 存储:
• 带有 16 位总线@1050Mbps 的 512MB DDR3。

• 16MB Quad-SPI 闪存，具有出厂编程的全球唯一标识符(兼容 48 位 EUI-48/64)。
• MicroSD 插槽。

(3) 供电：由 USB 或 7～15V 电源供电。

(4) USB 和以太网：

• 千兆以太网 PHY。
• USB-JTAG 编程电路。
• USB-UART 桥。
• USB OTG PHY(仅支持主机)。

(5) 音频和视频：

• 具有 24 位 DAC 且支持 I^2S 协议的 3.5mm TRRS 插孔。
• 3.5mm 线路输入插口。
• HDMI 接收端口(输入)。
• HDMI 源端口(输出)。

(6) 开关、按钮和 LED：

• 4 个按钮。
• 2 个滑动开关。
• 4 个 LED。
• 2 个 RGB LED。

(7) 扩展连接器：

• 两个标准 Pmod 端口。
• 16 个 FPGA I/O 接口(与树莓派接口共享 8 个引脚)。

(8) Arduino 屏蔽连接器：

• 24 个 FPGA I/O。
• 6 个 XADC 的单端 0～3.3V 模拟输入。
• Raspberry Pi 连接器。
• 28 个 FPGA I/O (与 Pmod A 接口共享 8 个)。
• 低延时控制。

从上面的介绍可见，PYNQ Z1 和 PYNQ Z2 都采用了同一型号的核心芯片，在开发板的配置上有很多的相似之处，因此对于计算机组成原理实验来说具体使用哪一款板卡都是可以的。

它们的主要区别在于扩展接头和音频系统。PYNQ-Z2 使用 Raspberry Pi 接头取代了 PYNQ-Z1 上的 chipKIT 接头。PYNQ-Z1 具有带 PWM 输入的集成 MIC 和单声道 PDM 音频输出。PYNQ-Z2 具有完整的 ADI 音频编解码器，还带有耳机、麦克风和线路输出。

1.2 Zynq 7020 芯片

从 1.1 节中的介绍可发现，PYNQ Z1 和 PYNQ Z2 开发板使用的核心芯片都为 Zynq 7020 芯片，具体型号为 Zynq XC7Z020-1CLG400C。

Zynq 7020 是 Xilinx 公司推出的全可编程 SoC 芯片（AP SoC，All Programmable System on Chips），与其他 Zynq 7000 系列芯片具有相同的架构，它主要由 PS（Processing System，处理器）和 PL（Programmable Logic，可编程逻辑 FPGA）两大部分组成，所以 PYNQ 可被看作一个具有可编程逻辑模块的计算机。其内部架构如图 1-3 所示。

Zynq XC7Z020-1CLG400C 芯片内的模块如下：

（1）主频 650MHz 的双核 ARM Cortex-A9 处理器。

（2）DDR3 内存控制器，具有 8 个 DMA 通道和 4 个高性能 AXI4 从端口。

（3）高速外设控制器：1GHz 以太网，USB 2.0，SDIO。

（4）低速外设控制器：SPI，UART，CAN，I^2C。

（5）可通过 JTAG，Quad-SPI 闪存和 MicroSD 卡进行可执行流文件的烧写。

（6）PL 部分采用的是 Artix-7 系列的可编程逻辑，具体资源如下：

- 13 300 个逻辑块，每个逻辑块具有 4 个 6 输入 LUT 和 8 个触发器。
- 630KB 的快速 Block RAM。

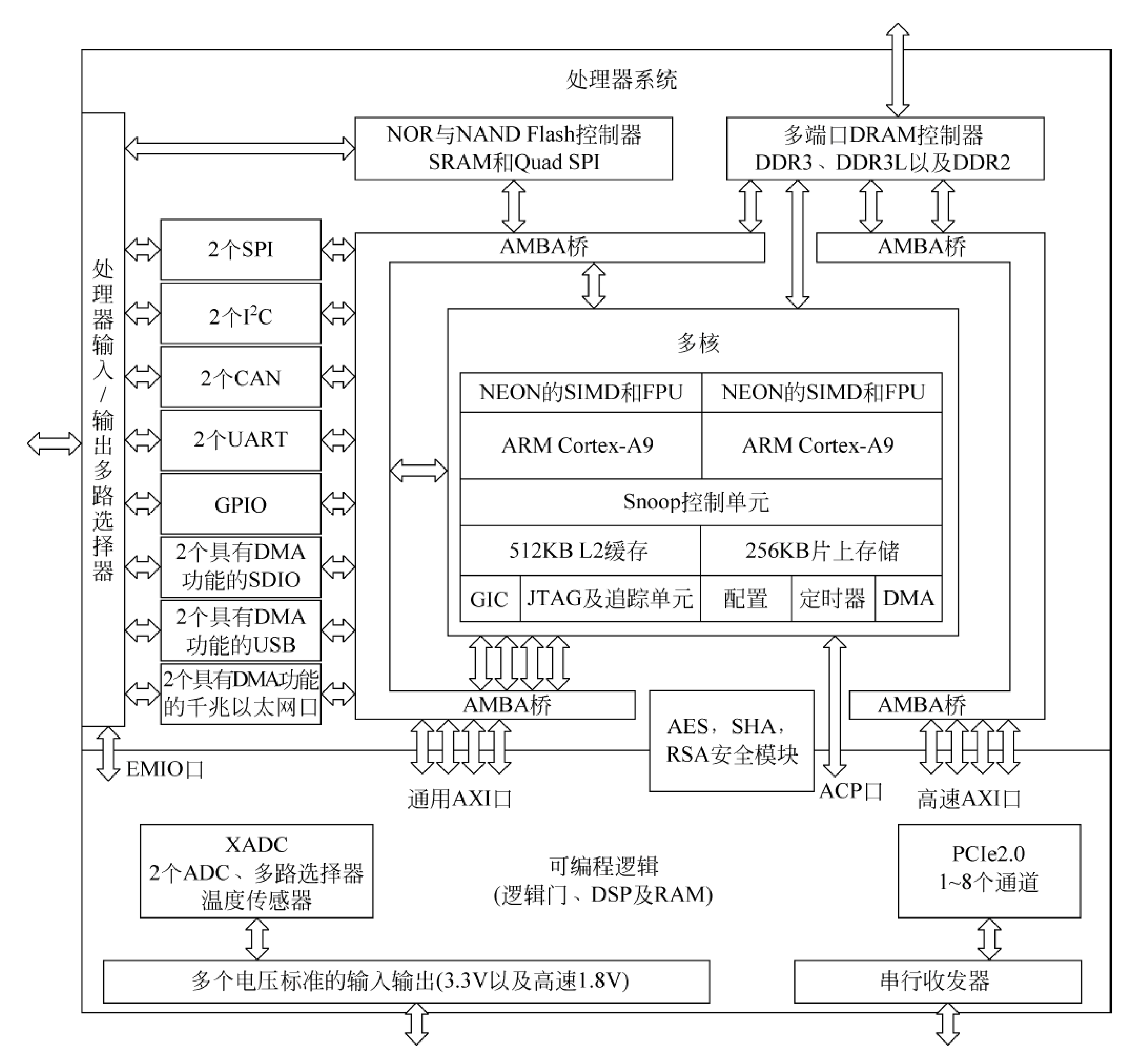


图 1-3 Zynq7000 芯片架构示意图

- 4 个时钟管理单元，每个有一个锁相环(PLL)和混合模式时钟管理器(MMCM)。
- 220 个 DSP。
- 片上模数转换器(XADC)。

1.3　PYNQ 框架

PYNQ 是一个开源框架，旨在使基于 Xilinx Zynq 的嵌入式系统设计更加容易。PYNQ 开源框架主要包含 PYNQ 硬件库(Overlay)以及 Overlay 的设计和调用方法。PYNQ 框架中实现了用来加载和使用 Overlay 的 Python 库，允许通过 PS 中运行的 Python 来控制 PL 中的 Overlay，Overlay 即在 PL 中运行的一个具体硬件模块实现。

因为 FPGA 设计通常需要具有硬件知识，PYNQ 的 Overlay 层由硬件设计人员创建，并提供相应的 PYNQ Python API。而软件开发人员就可以使用 Python 接口来调用和控制所需的硬件 Overlay 构成满足需求的软硬件协同计算系统。

Overlay 由三部分构成：

- 配置 PL 端 FPGA 的比特流文件(bit-stream)。
- 包含 IP 核信息的 Vivado Block Design TCL 文件。
- 读取 IP 核属性的 Python API。

PYNQ 框架的理念就是将软件和硬件开发隔离开，硬件人员仅关注特定 Overlay 的设计而无须关注整个应用，而软件或应用开发人员基于 Python 可快速开发适合不同需求的应用，并通过调用 Overlay 获得硬件加速来保证整个系统的运行效率。随着 Overlay 的不断积累和丰富，PYNQ 生态就会在应用的开发效率和系统的计算能效两方面都获得更好的表现。目前 PYNQ 官方支持的开发板有 Digilent 的 PYNQ Z1 开发板，TUL 的 PYNQ Z2 开发板及 Xilinx 自有的 ZCU104 开发板。此外，PYNQ 还可以支持安富利 Ultra96 及其他的一些第三方 Zynq 类的板卡。PYNQ 官网 http://www.pynq.io/board.html 提供了所支持的板卡的映像文件，同时介绍了如何为第三方板卡制作映像文件。

1.4　PYNQ 平台的使用

如前所述，PYNQ 平台就是一台由支持 Python 编程且具有可编程逻辑块的异构计算芯片构成的计算机。它可以放在现场使用，当然也可以部署在云端通过远程访问进行使用。关于 PYNQ 的实验环境的具体准备，详见第 2 章。

第 2 章 实验环境的准备

CHAPTER 2

实验环境的准备分为线下和线上两种方式：

(1) 线下方式指的是传统的在本地机器安装开发软件和连接实验板卡的方式。

(2) 线上方式指的是开发软件和实验板卡都部署在云上，用户通过远程登录网站访问软件和实验板卡的方式。

我们还可以将这两种方式组合使用：如本机安装 Vivado 软件，但到云上使用 PYNQ 板卡；或者 Vivado 在云上，但使用本地板卡。

2.1 线下方式实验环境的准备

2.1.1 在本机安装 Vivado 软件

访问 https://www.xilinx.com/support/download.html(Xilinx 官网)，下载 Vivado 在线安装文件(或者下载完整安装包)进行安装。以 Vivado 2018.2 为例(仅展示关键安装步骤)，在 Select Edition to Install 界面选择 Vivado HL Design Edition，如图 2-1 所示。

在安装成功后(见图 2-2)，打开 Vivado License Manager 2018.2，选择左侧的 Load License，然后单击 Copy License 按钮，选择 License，安装结束。

2.1.2 PYNQ 板卡的准备

1. PYNQ 板卡的启动

当用户以线下方式使用 PYNQ 时，首先要做的工作是准备一张至少 8GB 的 SD 卡，再在 PYNQ 官网下载与板子适配的 PYNQ 映像文件，并将映像文件烧录到 SD 卡中。PYNQ 开发板一般使用 USB 供电的方式，它同时可以作为串口使用。使用此种供电方式需要将图 2-3 中标号①处的电源跳线帽设置到 USB 位置，使用 USB 连接 PC 或笔记本电脑(后面统称本地计算机)和板卡。在启动 PYNQ 开发板时，首先将已经烧录好 PYNQ 映像文件的 SD 卡插入开发板卡槽中，然后通过图 2-3 中标号②处的 Boot 跳线帽选择从 SD 卡启动。打开电源开关，约 1min 后两个蓝色 LED 和四个黄色/绿色 LED 同时闪烁，随后蓝色 LED 灯熄灭，PYNQ 开发板启动成功。

2. PYNQ 板卡的具体连接

(1) 将 PYNQ 板卡和本地计算机直接相连。

- 使用网线将 PYNQ 板卡与本地计算机直连。

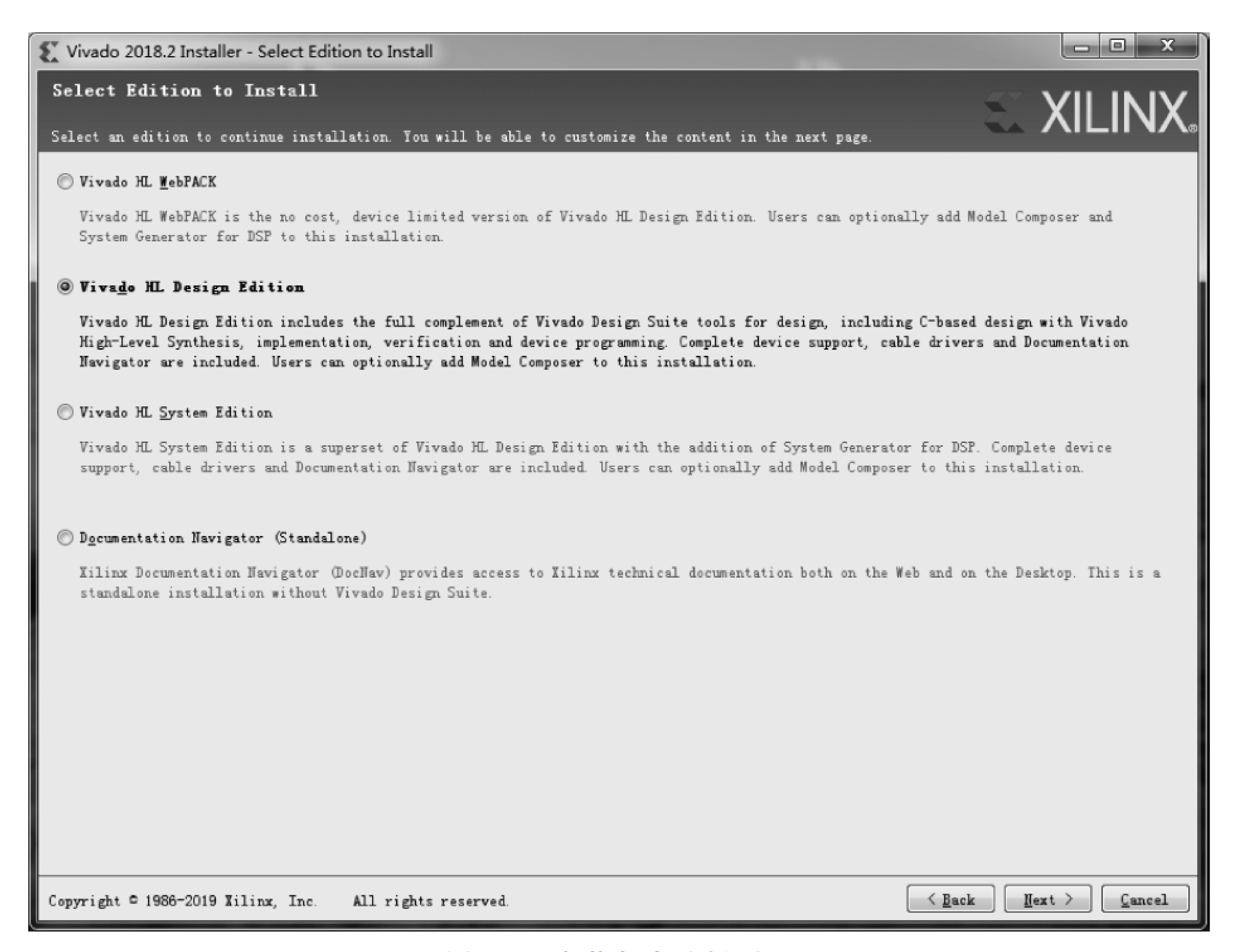


图 2-1　安装版本选择页面



图 2-2　安装证书

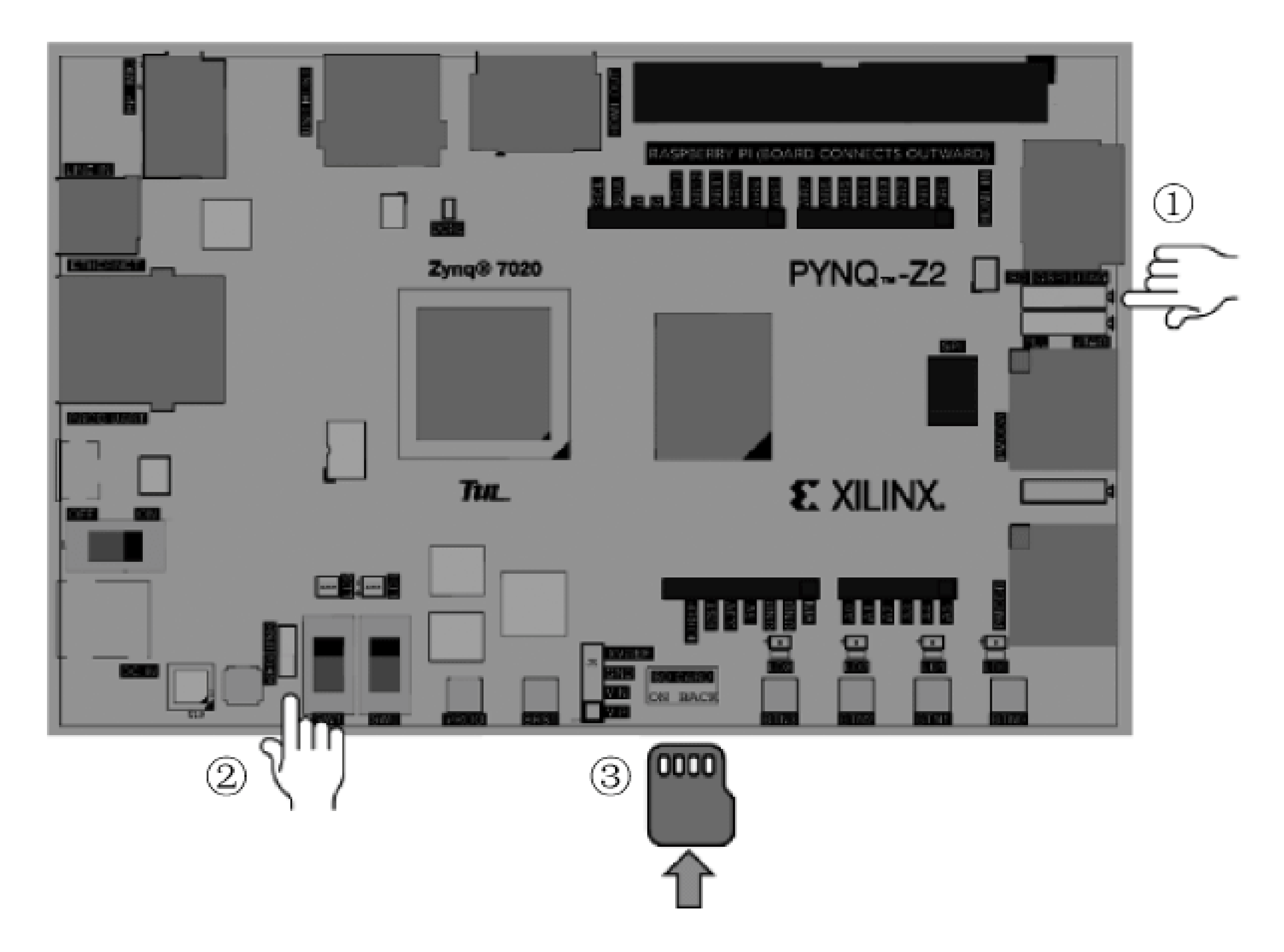


图 2-3　PYNQ 板卡的启动

- 查看 PYNQ 板卡 IP 地址。
- 根据 PYNQ 板卡 IP 地址来设置本机以太网静态 IP，保证本机 IP 和板卡 IP 在同一网段。例如，板卡地址为 192.168.2.99，那么可以设置本机 IP 为 192.168.2.1，掩码为 255.255.255.0。

直联的缺点是 PYNQ 板卡与本地计算机成为孤立的系统，无法访问到网络。

(2) 或者将 PYNQ 板卡连接到具有 Internet 访问权限的网络或者路由器端口，并使其与本地计算机处于同一网段，之后可以查看 PYNQ 板卡 IP 地址。

(3) 如何查看 PYNQ 板卡的 IP 地址。

PYNQ 板卡启动后，可以通过串口终端软件 Xshell 或者开源免费的 Putty 查看板卡 IP。以 Putty 为例：打开 Putty，选择 Serial。其中，COM 端口可以在“设备管理器”下的“端口”中查到。填写 COM 端口和 Speed(如 115200)，再单击 Open 按钮，如图 2-4 所示。使用命令 ifconfig 可以查看板卡 IP 地址(方框选中的内容)，如图 2-5 所示。

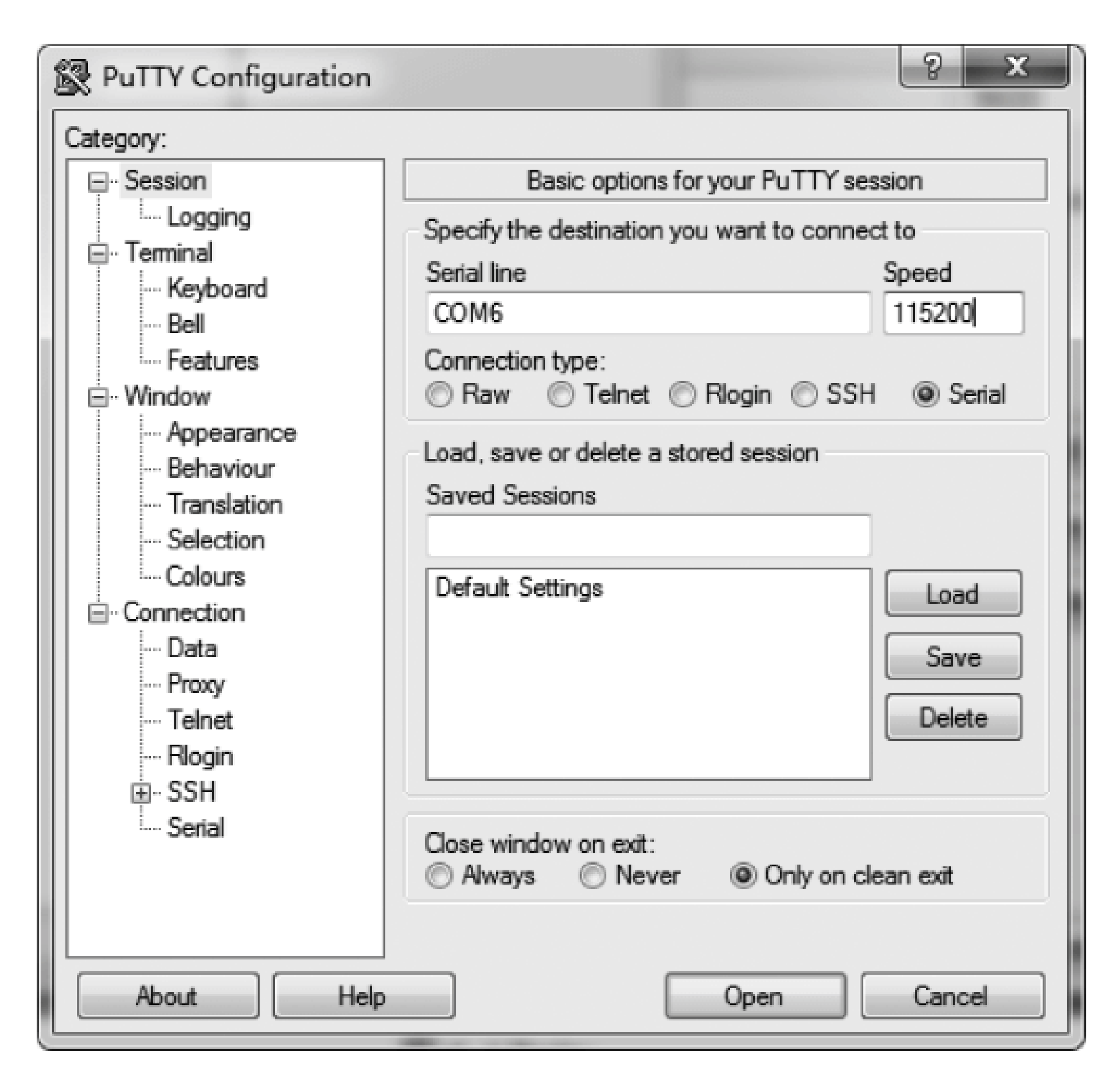


图 2-4　Putty 界面

```
xilinx@pynq:~$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.18.139.201  netmask 255.255.255.0  broadcast 172.18.139.255
        inet6 2001:da8:1010:803b:205:6bff:fe00:b5ec  prefixlen 64  scopeid 0x0<g
lobal>
        inet6 fe80::205:6bff:fe00:b5ec  prefixlen 64  scopeid 0x20<link>
        ether 00:05:6b:00:b5:ec  txqueuelen 1000  (Ethernet)
        RX packets 1778  bytes 287283 (287.2 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 573  bytes 63593 (63.5 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 29  base 0xb000

eth0:1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.2.99  netmask 255.255.255.0  broadcast 192.168.2.255
        ether 00:05:6b:00:b5:ec  txqueuelen 1000  (Ethernet)
        device interrupt 29  base 0xb000

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 107  bytes 15496 (15.4 KB)
```

图 2-5　查看板卡 IP

2.1.3　使用 Jupyter Notebook 与 PYNQ 建立连接

打开 web 浏览器(建议使用 Chrome 或 360 极速浏览器),访问板卡 IP 地址就连接到了 Jupyter Notebook 服务。此时需要输入用户名和密码,请输入用户名:xilinx,密码:xilinx,即可打开如图 2-6 所示的 Jupyter Notebook。单击右上角的 Upload,选择想要上传的文件,单击“打开”按钮,出现如图 2-7 所示的界面,再单击 Upload 即可上传。Upload 是把在本机的文件上传到了 PYNQ 板卡的文件系统,供后续在 Jupyter Notebook 中使用。



图 2-6　Jupyter 界面

图 2-7　上传文件

另外,还可以单击某一个文件左侧的方框,选择 Download 将 PYNQ 板卡的文件下载到本地计算机上。注意,下载的 xxx.ipynb 文件必须是非运行状态的(表现为灰色图标),如图 2-8 所示。如果要下载的文件刚运行过,那么可以选中该文件选择 Duplicate 将其复制,

复制的文件是非运行状态，下载复制文件即可。



图 2-8　下载文件

2.2　线上方式实验环境的准备

OpenHEC 是一个 FPGA 云平台系统，通过在数据中心部署 Vivado 虚拟机及 PYNQ 开发板，使用户无须安装软件并购买 FPGA 板卡就可以进行实验。具体使用方式如下：

(1) 输入网址并登录，如果没有注册，先进行注册：

- 公有云：www.iopenhec.com。
- 私有云：如果部署有私有云，则输入相应的网址或 IP 地址。

为了展示方便起见，以 OpenHEC 公有云为例说明。输入网址 www.iopenhec.com 进入主页，根据指示可访问相应的 PYNQ 节点，里面可以看到“使用虚拟机”和“使用 FPGA”两个选项。

- 使用 Vivado 软件：单击“使用虚拟机”，进入后，在桌面上双击 Vivado 打开使用，如图 2-9 所示。



图 2-9　使用 Vivado

- 使用 PYNQ 板卡：单击“使用 FPGA”，可直接连接到 Jupyter Notebook 服务并通过输入 xilinx 打开。如图 2-10 所示，如果没有直接连接，则打开 Chronium Web

Browser，输入网址 pynq：9090，打开后再输入密码 xilinx，之后的使用方式和线下一样。



图 2-10　使用 FPGA

（2）oDisk 的使用：在使用线上方式，或者线上线下结合的方式时需要注意，OpenHEC 系统中提供了一个共享云盘（oDisk）（见图 2-11），用来在不同的虚拟机空间，以及用户的本机之间传递文件。如果都使用线上的方式，用户在使用 Vivado 和使用 FPGA 时，可以共享 oDisk 里面的文件。如果线上线下结合使用，则可以通过 OpenHEC 提供的文件上传、下载到 oDisk 和本机之间进行文件交换。



图 2-11　oDisk 共享云盘

第 3 章

CHAPTER 3

基于 PYNQ 的组成原理实验流程概览

本章主要介绍本书所用的 FPGA 实验平台(PYNQ)以及在此平台上进行组成原理实验的大致流程,目的是为读者建立一个整体的概念,方便读者对后续章节内容的理解。

3.1 整体开发流程介绍

如图 3-1 所示,当读者进行组成原理相关的实验时,例如设计一个运算器(ALU),通常可以采用原理图或硬件描述语言(Verilog/VHDL)作为设计输入方式,并使用 Xilinx 提供的数字系统设计软件 Vivado 进行具体的设计。在使用 Vivado 进行开发的过程中,还需要指定所用的 FPGA 开发板或 FPGA 芯片。

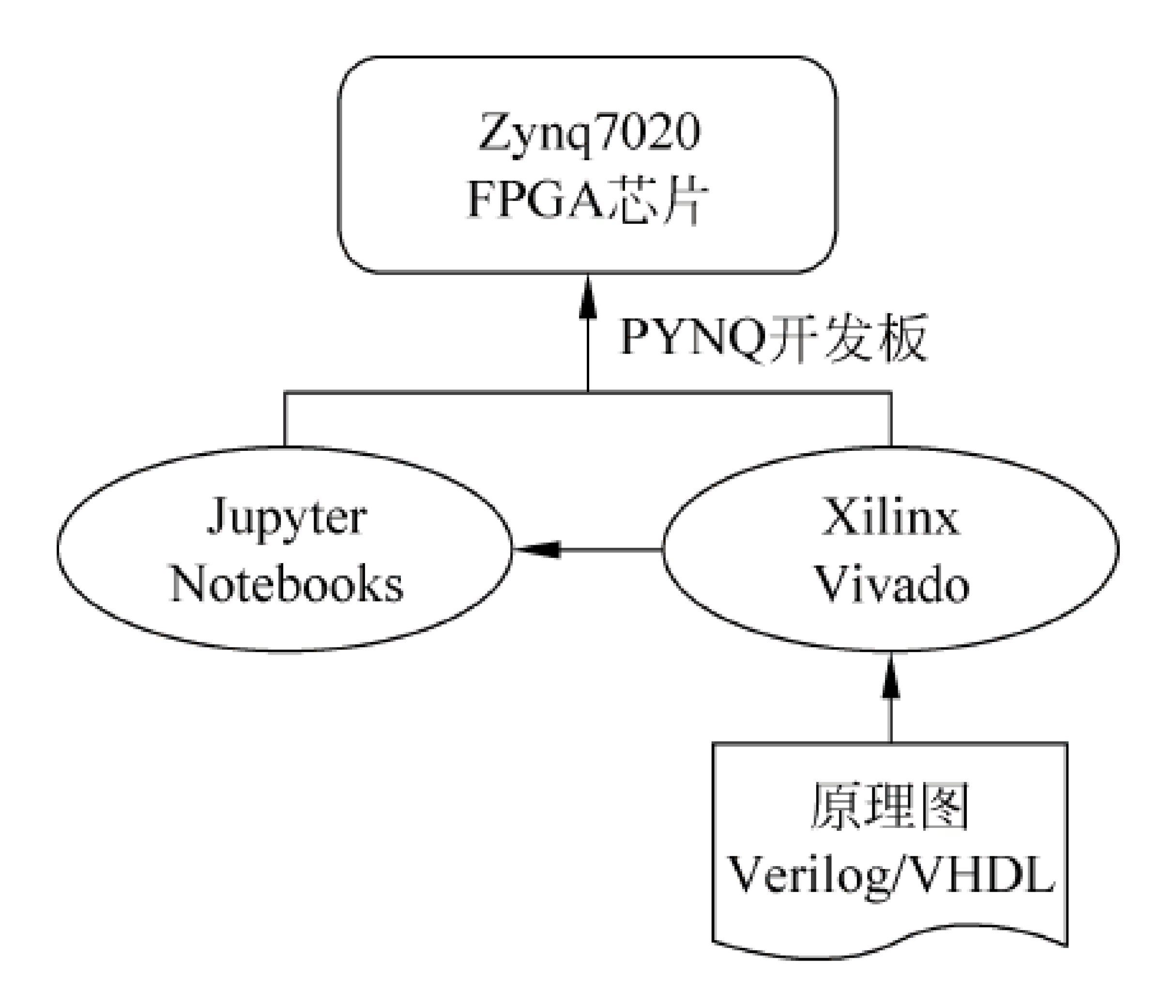


图 3-1 基于 PYNQ 的组成原理实验流程概览图

如图 3-1 所示,PYNQ 开发板就是本书所用的 FPGA 开发板,其所使用的 FPGA 芯片是 Zynq 7000 系列的 Zynq 7020,具体型号为 xc7z020clg400-1。

在使用 Vivado 经过设计输入完成硬件模块设计之后,再经过仿真、综合、引脚绑定、布局布线等流程之后,就可以生成 FPGA 芯片(对于 PYNQ 来说,就是其芯片的 PL 端)上运行的文件,通常称为流文件或 bit 文件(.bit 或.bin 文件)或比特流。虽然使用 Vivado 可以直接将流文件下载到 FPGA 上,但在基于 PYNQ 的开发中,当流文件生成之后并没有直接进行下载,而是采用图 3-1 所示的 Jupyter Notebook 进行 Python 代码的编写,并在 Python 代码中完成流文件的下载。在 Python 代码中还可以完成和流文件所实现的硬件系统的交互。

如第 2 章所述,进行组成原理实验时需要用到 Vivado 开发工具及 PYNQ FPGA 实验板卡,这些开发环境的提供可以采用线上的方式(即 Vivado 和 PYNQ 可以部署在云上)也可以采用线下的方式(即 Vivado 和 PYNQ 部署在本地机器),甚至线上线下混合的方式。笔者认为,较为理想的方式应该是线上线下结合的实验方式。课内实验建议采用线下方式,让学生对真实硬件有更直观的感受;课外则要能更广泛地支持在线实验,使学生可以随时随地进行实验,从而为学生提供可与软件实验相比的实验条件。

需要注意的是,Zynq7020 芯片是一款 SoC 芯片(具体已在第 1 章中详细介绍),它主要由双核的 ARM 处理器(在 Zynq 中通常称为 PS,Processing System)和 FPGA(通常称为 PL,Programmable Logic)两部分构成。由 Vivado 生成的流文件运行在 PL 上,而 Jupyter Notebook 及 Python 运行在 PS 上,PL 和 PS 之间通过总线进行连接,两者共同构成了一个软硬件协作的系统。

3.2　Vivado 开发流程概览

本节仅介绍 Vivado 开发流程的概要,目的是从整体上提炼 Vivado 的功能和使用方法,让读者建立全局的认识,而不至于一开始就陷入工具使用的细节,影响对整体的理解。更详细的 Vivado 使用细节将在第 4 章中进行介绍。

1. 设计输入

设计输入是指用什么样的方式进行系统的具体设计。通常用于设计输入的方式有两种:原理图的方式和硬件描述语言的方式。

采用原理图的方式比较直观,易于理解硬件结构,且不需要进行专门的学习,比较适用于初学者及复杂程度不是很高的系统。

硬件描述语言方式,即 Verilog HDL 或 VHDL,是通过编写代码的方式设计硬件系统,在描述复杂程度较高的系统时效率比较高。因此在实际的工程实践中,通常采用硬件描述语言的方式。但其不足之处是需要提前进行一定的专门学习才能掌握,具有一定的使用门槛。

当然,两种设计输入方式可以混合使用。用户采用 Vivado 进行系统设计时,首先要创建一个工程,然后可以为该工程添加或创建设计输入文件,即可开始进行具体的设计输入了。

2. 仿真

仿真是指不把设计的硬件系统直接下载到芯片上真实运行,而是通过 Vivado 软件对其进行分析(实际上是模拟)。仿真具体又分为前仿真和后仿真。

前仿真,也称为功能仿真或行为级仿真。是指仅对逻辑功能进行测试模拟,以了解其实现的功能是否满足原设计的要求。仿真过程没有加入时序信息,不涉及具体器件的硬件特性,如延时特性。在组成原理的实验过程中应用频率较高。

后仿真,也称为布局布线后仿真或时序仿真。是指提取有关的器件延时、连线延时等时序参数,并在此基础上进行的仿真,它是非常接近真实器件运行情况的仿真。

基于 FPGA 的系统设计中,通常有 2 个阶段可以进行后仿真:第一个阶段是针对特定的 FPGA 厂家技术的仿真,此仿真是在综合之后、实现之前进行的仿真。另一个阶段是门级仿真,此级仿真是针对实现后的门级时序进行仿真,门级仿真体现了由于布局布线而产生的实际延时。

3. 综合

综合的功能有点类似于编译,编译所做的工作是从用户输入的源代码生成特定的机器代码。综合所做的工作是从用户的输入生成特定的网表文件。综合具体包含语言的编译、结构的优化及面向芯片的映射和转换。

4. 实现

实现主要是完成转换、映射、布局、布线和最终比特流的生成。意思是将综合生成的网表一步步地与特定的FPGA芯片关联起来，转换成该FPGA芯片上特有的资源并进行映射。布局指的是根据网表在FPGA芯片上定位所涉及的特定资源的位置；而布线指的是这些特定资源之间如何连接起来。在布局、布线之后还可以进行进一步的仿真，然后就可以生成在特定FPGA芯片上运行的流文件了。

在综合之后、实现之前，还有个重要的工作是FPGA芯片引脚的绑定。引脚绑定的主要目的是将用户设计的硬件系统的逻辑I/O与FPGA的物理I/O根据需要连接起来，从而可以通过FPGA开发板上的I/O部件（如拨码开关、LED灯、七段码等）与用户设计的硬件系统进行交互，使得用户设计的硬件系统成为一个具备I/O的完整的系统。

需要注意的是：在基于PYNQ的组成原理实验中，由于Zynq 7020芯片是SoC芯片，Python代码可以运行在PS端并和PL端的用户模块进行交互。因此，在该平台上的引脚绑定阶段，可以通过将用户设计的硬件系统的I/O与PS端相连接，从而可以与PS端运行的Python进行交互。

3.3 基于Python的硬件交互

众所周知，目前在基于FPGA进行的数字逻辑、组成原理等硬件类课程的实验教学中，通常采用线下实验平台的方式。完成系统设计后，通过实验平台提供的物理I/O与系统进行交互来进行系统的验证。由于物理I/O的数量通常有限，且不能根据需要进行方便地调整，限制了系统调试的便利性。而目前采用线上FPGA平台的方式进行组成原理实验教学时，更多的做法还是将线下的实验方式搬到线上来，通过摄像头监控的方式或者将远程的FPGA平台以图形的方式展示在Web界面上。用户通过Web页面的虚拟拨码开关、LED等方式与远程的FPGA平台进行交互。这种方式的优点是直观、真实感强，学生能感觉到是在使用远程的FPGA平台进行实验。但由于物理I/O的局限性，无法很好地满足不同实验的调试需求。同时，很难平滑地延伸到对深度学习加速器设计等非冯·诺依曼架构的实验教学中，因为此类实验通常需要更复杂的I/O。

随着人工智能和大数据的发展，Python语言已发展成为深度学习人工智能领域最炙手可热的编程语言，越来越多的学校和机构开始开设Python语言。本书介绍采用Python语言进行组成原理实验中所需的I/O功能，一方面可以提升I/O的灵活性，丰富调试手段；另一方面，利于学生理解软硬件协同机制，并便于进一步扩展到深度学习加速器设计等非冯·诺依曼架构的实验教学。

如3.2节最后一段所述，Python代码可以运行在PS端并和PL端的用户硬件模块进行交互。Python代码可以灵活的方式进行不同形式的输入、输出工作，从而满足不同的硬件系统需求。

第 4 章 Vivado 开发流程

CHAPTER 4

Vivado 是全球最大的 FPGA 芯片提供商 Xilinx 于 2012 年发布的集成设计环境。与上一代开发工具 ISE 相比，Vivado 更加强调对软件硬件全可编程的支持。从 Xilinx 最新推出的 7 系列芯片（主要包括 Artix-7、Kintex-7 和 Virtex 三个子系列）开始，ISE 将不再提供支持，而需要采用 Vivado 开发软件。在采用 Vivado 进行组成原理实验时，硬件模块或系统的设计流程如下所述。

4.1 创建工程

在 Vivado 中设计一个自己的硬件模块或系统的第一步就是要创建一个工程。工程用来组织用户开发中的所有要素：如源文件、IP、TCL 等。后续的整个开发流程就是基于所创建的工程开展的。

下面的开发流程是以 Vivado 2018.2 为例，其余版本的 Vivado 使用与其差异也很小。创建工程首先需要提供工程名与工程路径，然后指定用到的工程类型，最后指定工程资源文件并指定工程所用的具体 FPGA 器件。

1. 启动 Vivado 2018.2

如图 4-1 所示为 Vivado 启动界面。

2. 选择 Create Project 新建工程

选择 Create Project 新建工程，进入到图 4-2 所示的新建工程向导界面。在此为新建的工程命名，并指定工程在计算机上的位置路径。注意路径中不要使用中文字符，且路径长度不要超过 260 个字节。

工程路径默认勾选 Create project subdirectory 用于创建工程子目录，勾选后会创建一个文件名与工程名相同的文件夹，用于存放整个工程。如果选择不勾选，将在当前文件夹下创建工程。

3. 工程类型选择

填写完文件名及路径后，单击 Next 按钮，进入到图 4-3 所示的工程类型选择界面。常规的工程选择为 RTL Project。Do not specify sources at this time 选项用来选择在新建工程过程中是否指定已有的源文件。如果勾选此项，则在此处进行工程所需的现有的源文件的指定，将会连续弹出新建或添加已存在的源文件、添加已存在的 IP、新建或添加已存在的

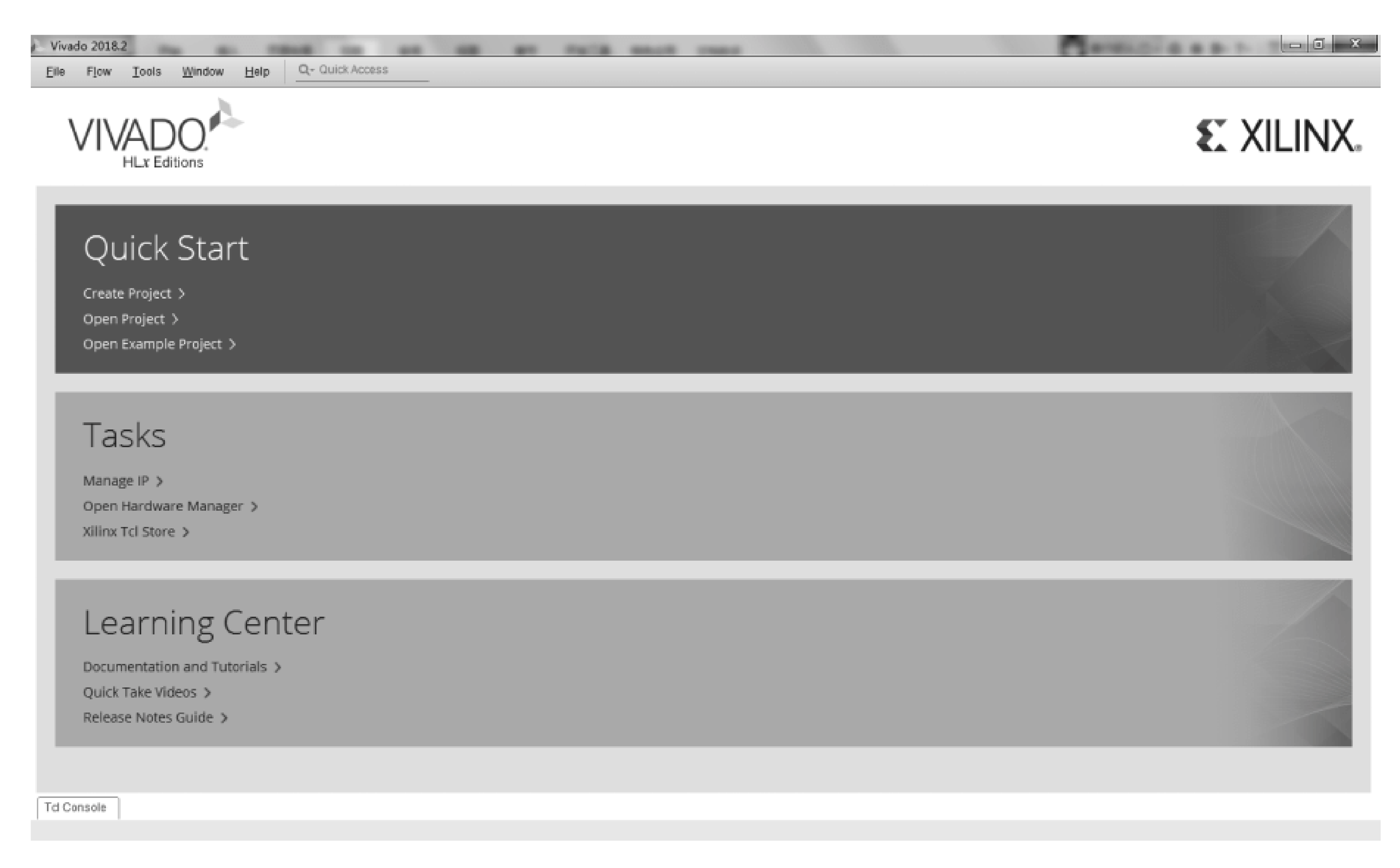


图 4-1 Vivado 启动界面

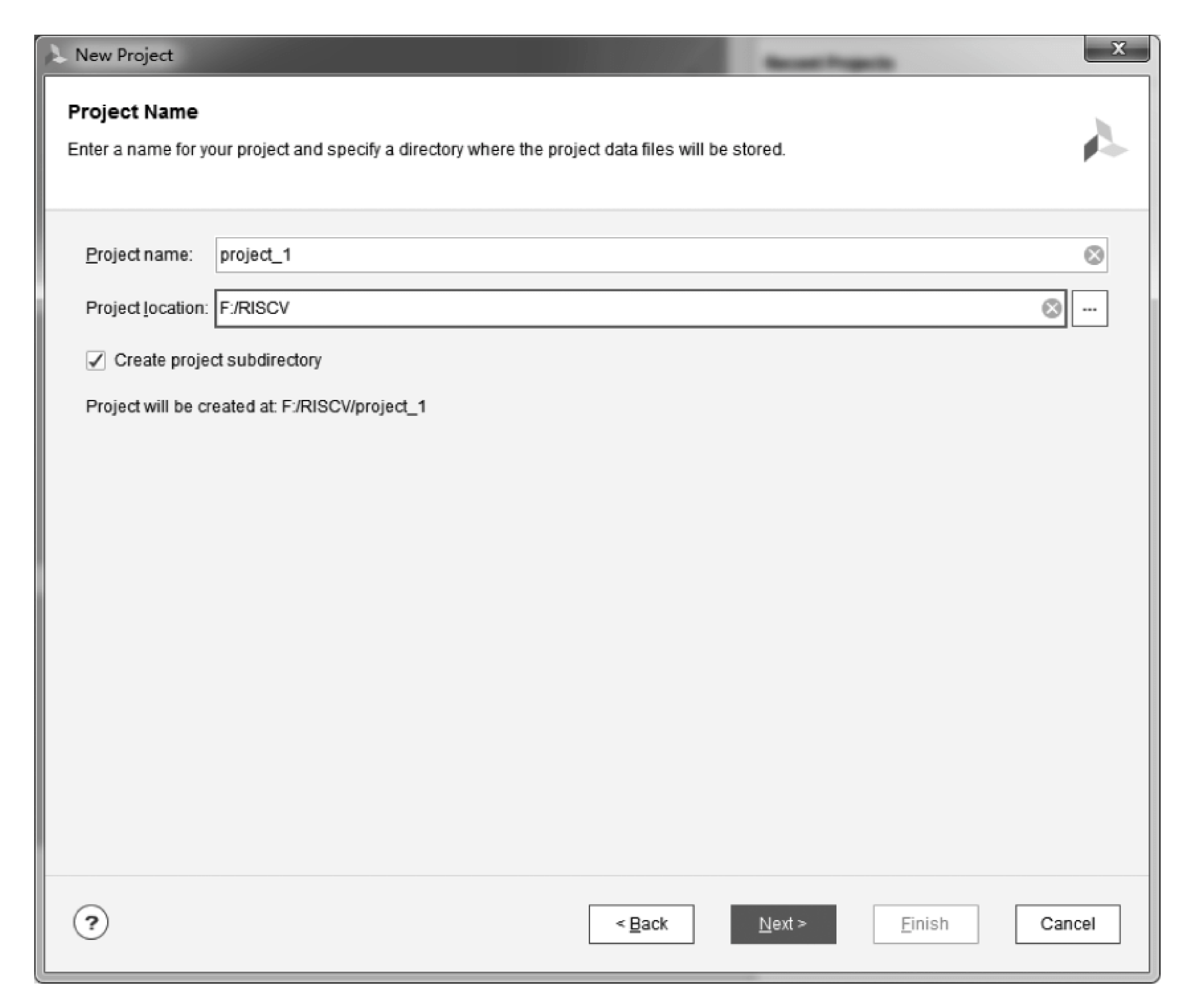


图 4-2 新建工程向导界面

约束文件。如果不勾选，则将源文件的添加或创建活动推迟到工程建好之后。在此以不需要新建或添加任何文件为例，所以这里选择 Do not specify sources at this time。在 RTL 工程建好之后，可以再新建添加源文件、创建 Block Design、生成 IP 核，运行 RTL 分析、综合、实现、布局布线。

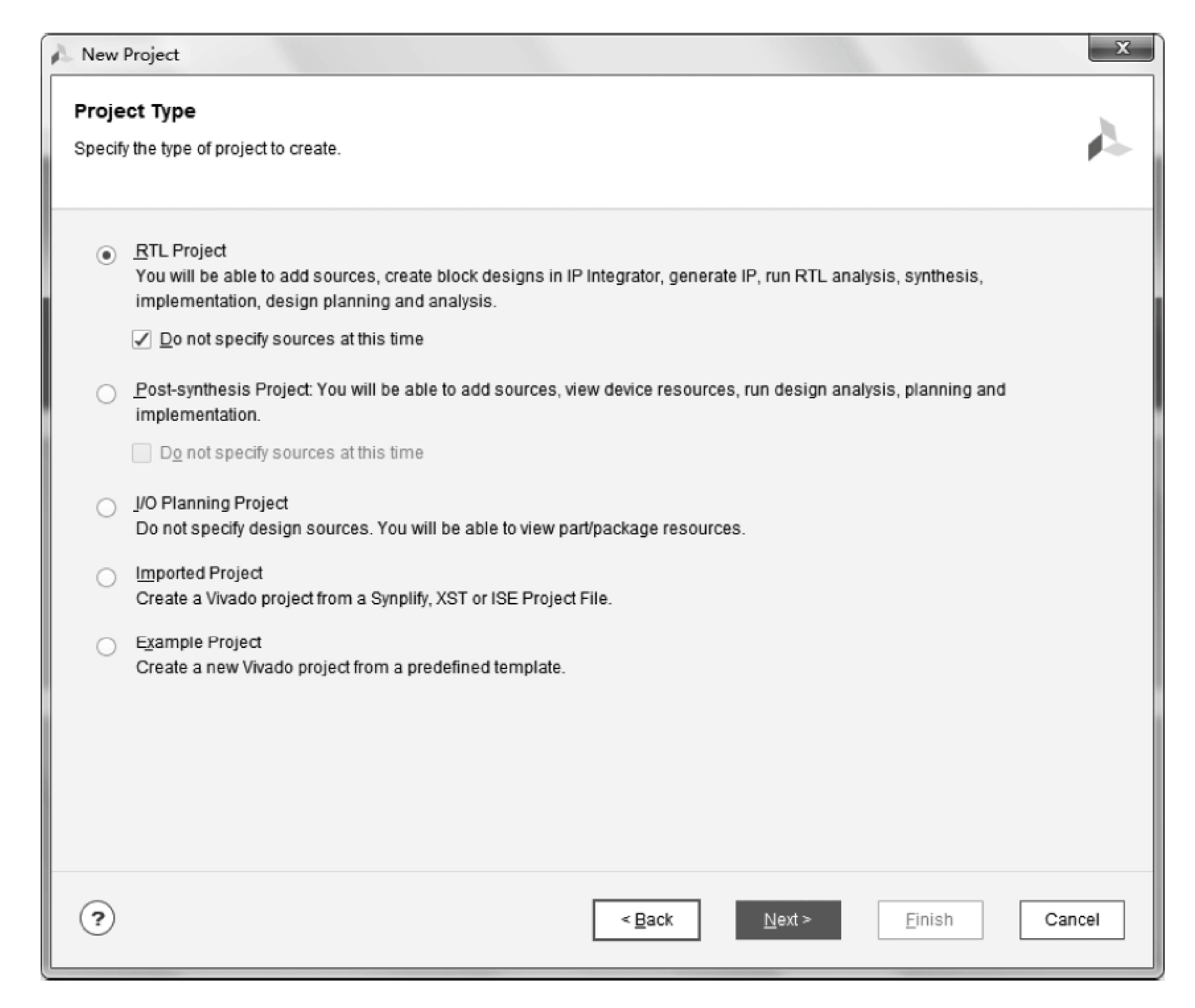


图 4-3　工程类型选择界面

4. 指定所用的 FPGA 器件或开发板

在图 4-3 所示的界面上单击 Next 按钮即进入图 4-4 所示的界面，在此处进行本工程所用的 FPGA 器件的选择，所选的器件后续还可以进行更改。

此处的 Parts 指的就是 FPGA 芯片，包含了 Vivado 2018.2 所支持的芯片型号。而 Boards 指的是 Vivado 2018.2 所支持的开发板。如 PYNQ Z1/Z2 开发板所用的芯片型号为 ZYNQ XC7Z020-1CLG400C，所以可以直接搜索 xc7z020clg400-1 添加此器件。添加 Parts 和 Boards 的区别是：直接添加 Parts 后还需自行添加 PYNQ 引脚的配置文件，而添加 Boards 后不需要。为方便起见，这里直接选择 Boards。

需要注意的是，由于 Vivado 中并不能直接选择到 PYNQ，因此要在 Boards 中选择 PYNQ Z1/Z2 时还需要自行添加 PYNQ 的支持文件 Boardfiles。从 Github：https://github.com/xupsh/pynq-supported-board-file 处可以下载 Boardfiles，解压后将 PYNQ Z1/Z2 复制到 Vivado 所安装的路径下，如 D:\Xilinx\Vivado\2018.2\data\boards\board_files（此处 Vivado 安装在 D 盘）中。重启 Vivado，在 Boards 列表中就会看到 PYNQ Z1/Z2 板卡信息，如图 4-5 所示。

5. 完成工程创建

从图 4-5 处选择 pynq-z2，单击 Next 按钮进入图 4-6，显示了所创建的工程的相关信息。单击 Finish 按钮即完成整个工程的创建。

6. 进入工程界面

工程创建完成后，主界面如图 4-7 所示。左侧面板为设计流程的管理，工程的源文件添加、综合、仿真、布局布线、生成比特流等操作都包含在设计流程管理中。中间面板为工程源

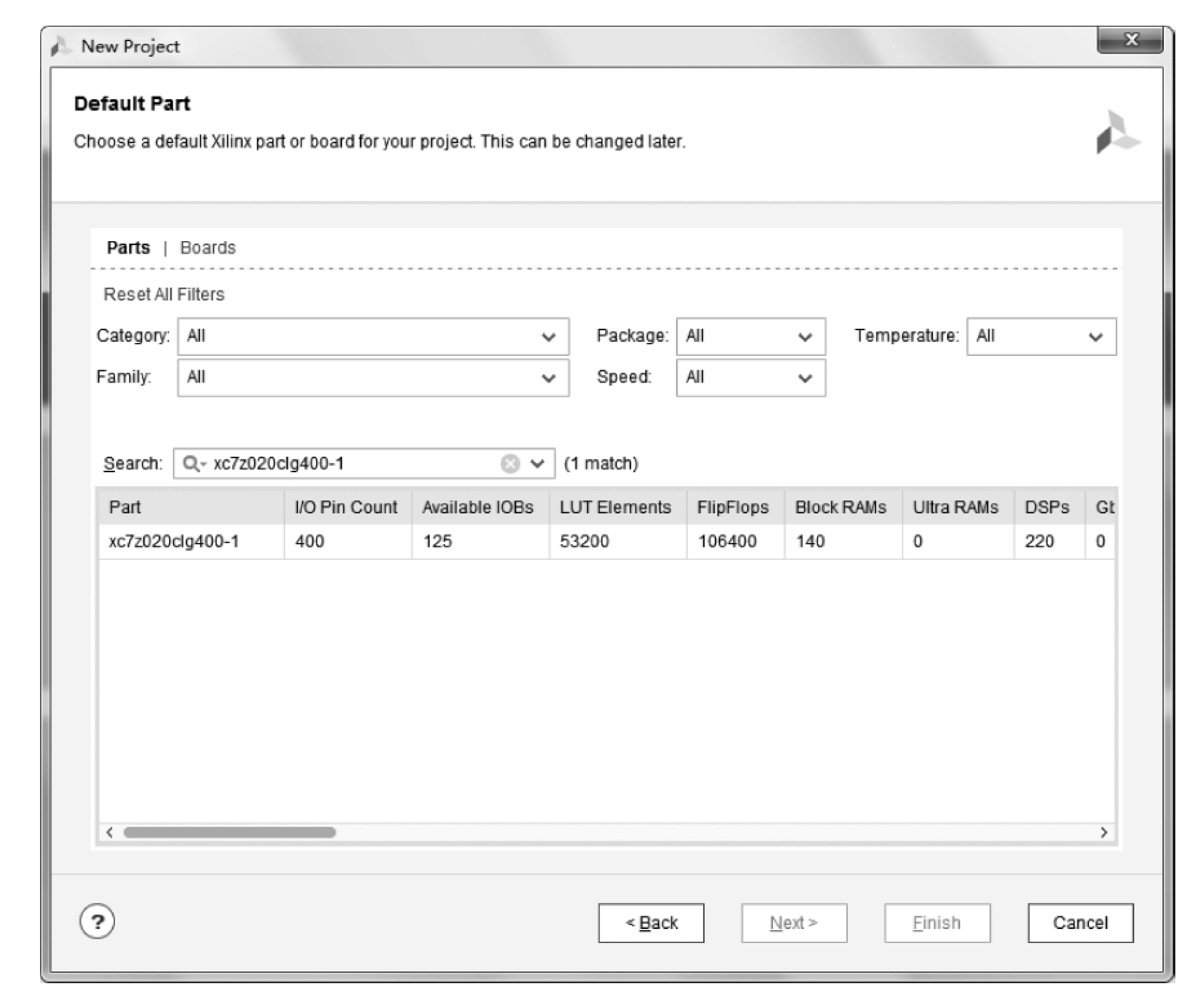


图 4-4 Parts/Boards 的选择界面

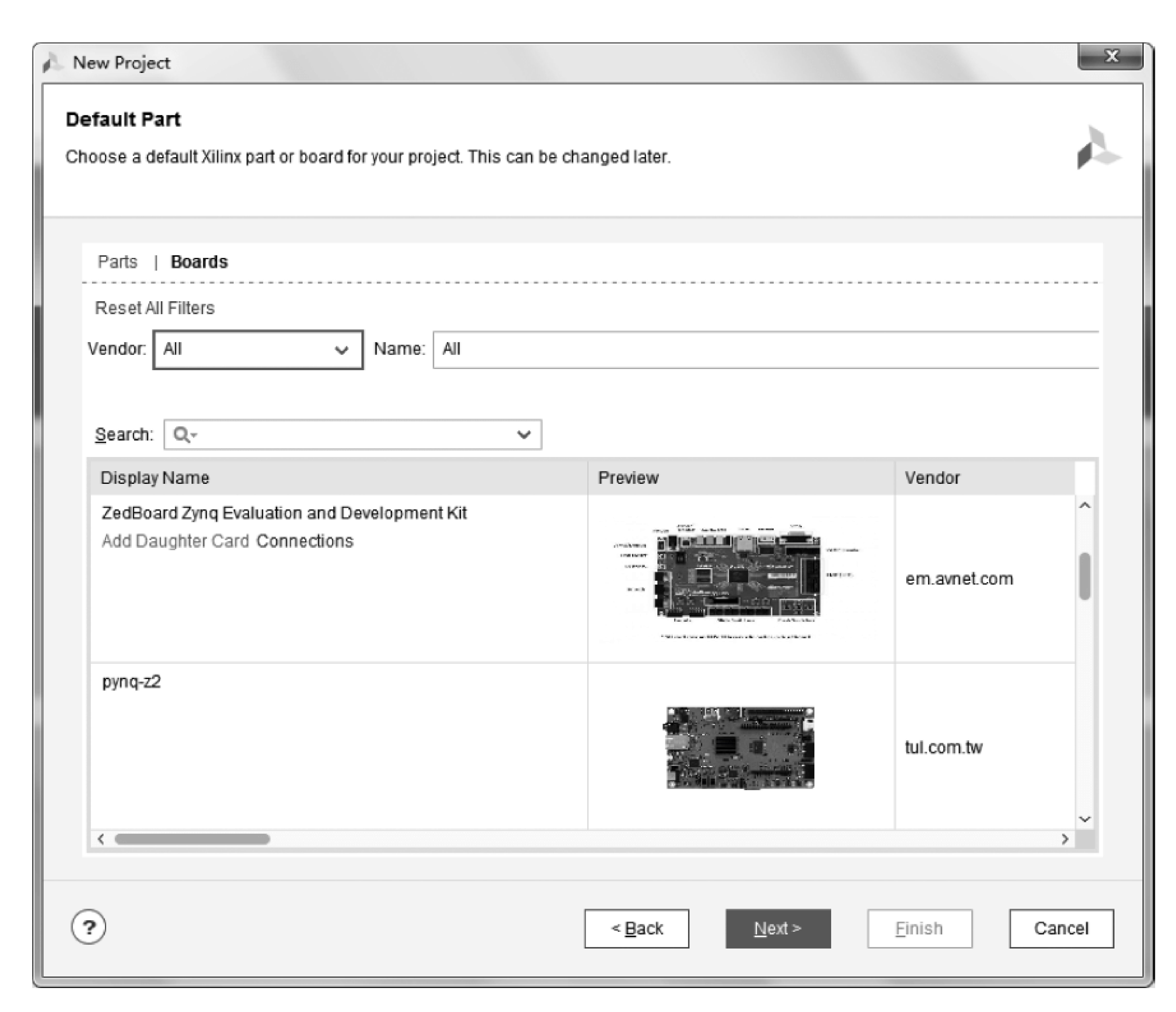


图 4-5 Boards 选择列表

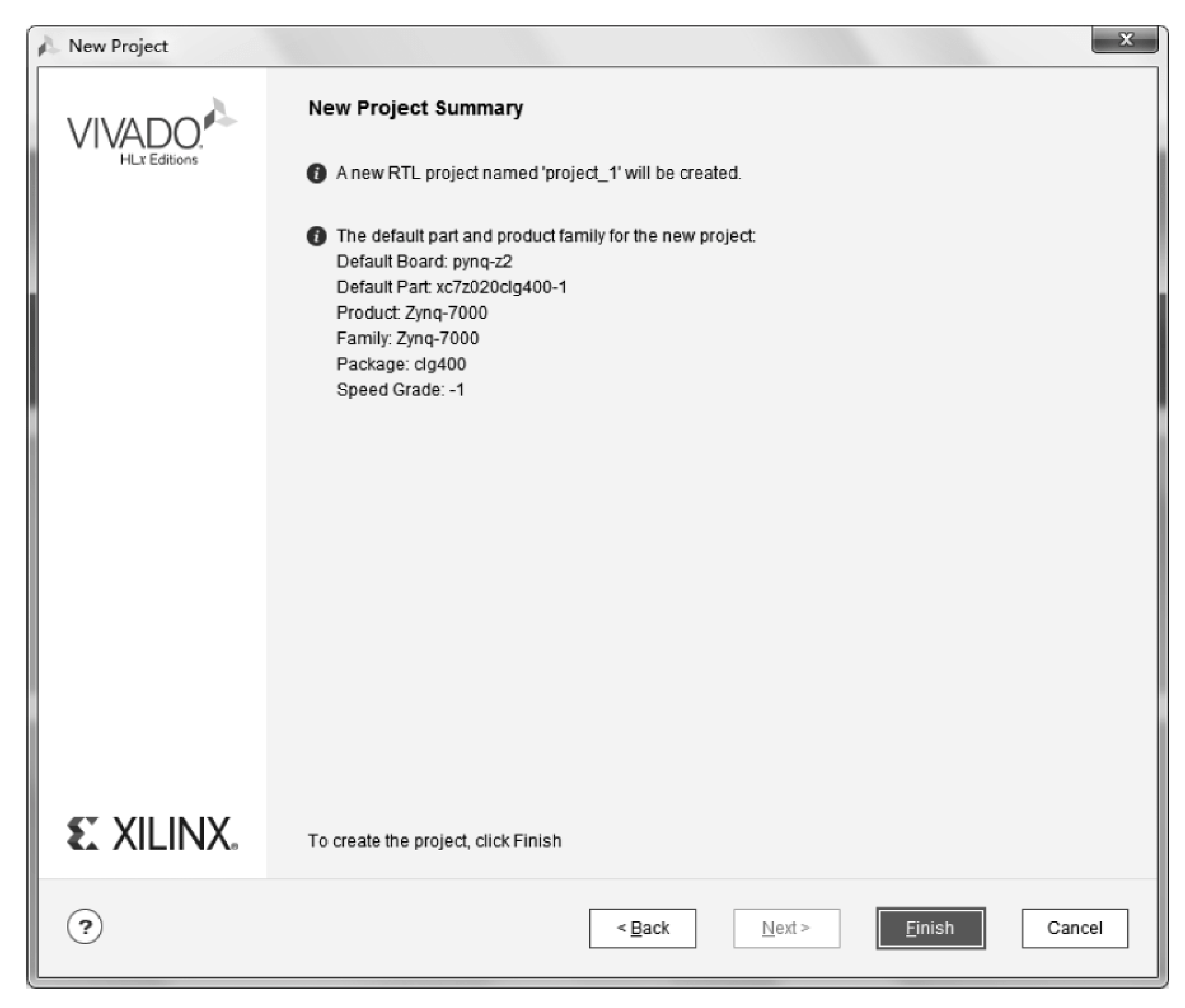


图 4-6　所创建的工程信息页面

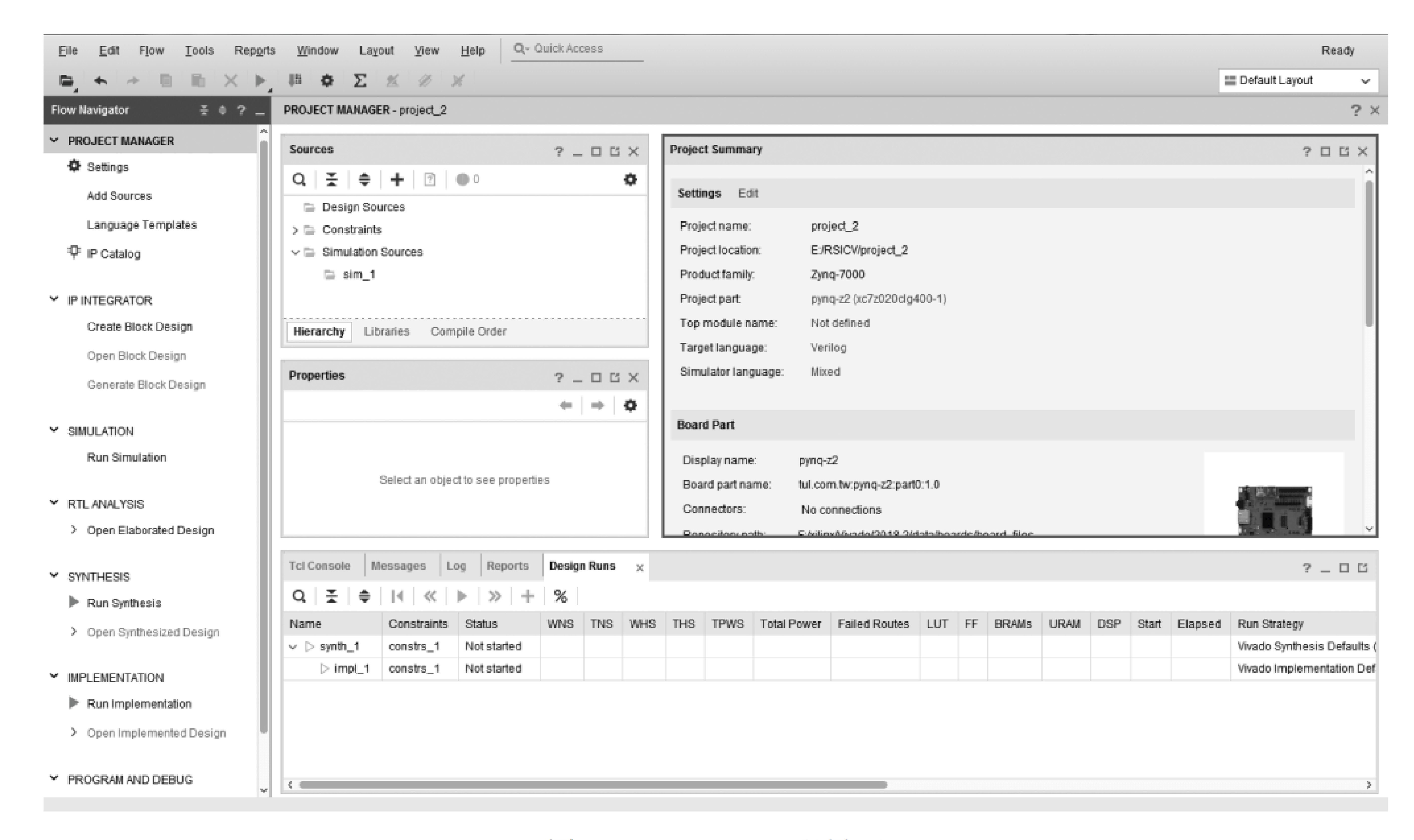


图 4-7　Vivado 工程界面

文件,包含设计源文件、引脚约束文件、仿真文件的选择面板。右侧面板为具体操作界面,以及工程信息,包含板卡信息、FPGA 资源利用率等概要信息。下部的面板为各种信息的输出,包含编译信息、报告等。

4.2 设计输入

设计输入就是用户采用什么样的方式进行具体开发，在计算机组成原理实验的开发中，通常可以采用原理图或 Verilog/VHDL 的方式进行设计输入。

4.2.1 原理图方式

所谓原理图方式，就是通过画电路图的方式非常直观地设计一个功能模块或系统。在 Vivado 里面可以使用图 4-7 工程界面左侧的 IP Integrator 下的 Block Design 进入原理图设计。采用原理图方式开发时，可以添加各种 IP 核（已有的功能模块），再进行连线配置。使用图形化的开发方式，就像在 FPGA 上搭建积木一样，非常直观方便。

下面以 Vivado 自带的 IP 核加减法器（Adder/Subtracter IP）为例，介绍 Vivado 的原理图开发方式。

在 Vivado 工程新建完成后，单击左侧 Create Block Design 新建原理图，进入如图 4-8 所示的界面。

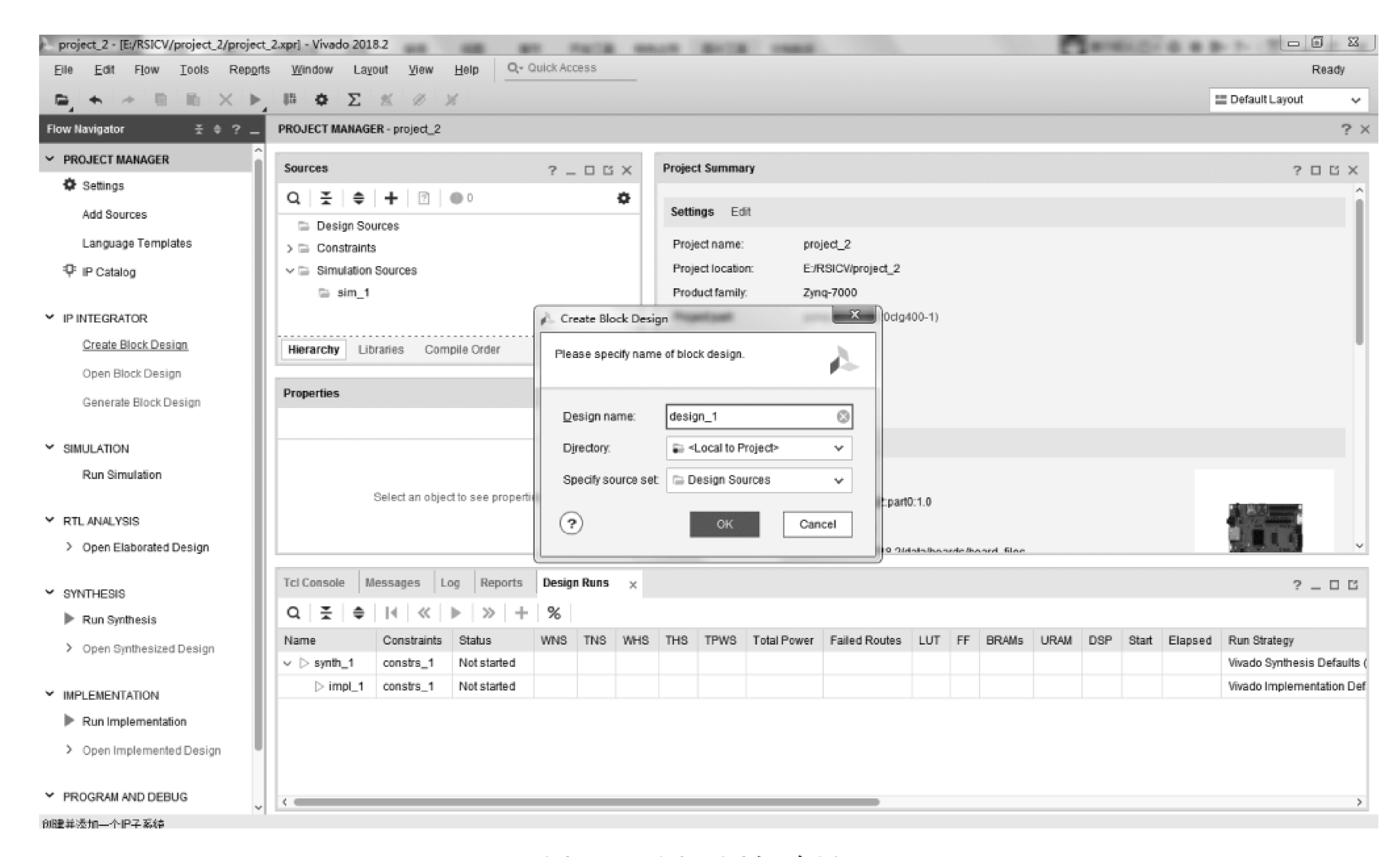


图 4-8 原理图新建界面

为该原理图命名后，单击 OK 按钮进入图 4-9。在图 4-9 中的 Diagram 中右击，选择 Add IP 添加 IP 核，搜索想要添加的 IP 模块名字 Adder/Subtracter。双击进行确认，添加加法器/减法器 IP 核，即成功添加了加减法器的 IP 核，如图 4-10 所示。

双击 Adder/Subtracter IP 核或者右击，选择 Customize Block，可以对 IP 核进行配置，如图 4-11 所示。用户可以根据需求调整 IP 的具体参数，例如可选择使用 FPGA 逻辑单元还是使用 DSP48 单元进行加减法的运算，并通过 Information 查看 IP 核占用的资源估算。在 IP 核配置窗口，单击左上角 Document 可查看与 IP 核相关的技术手册。

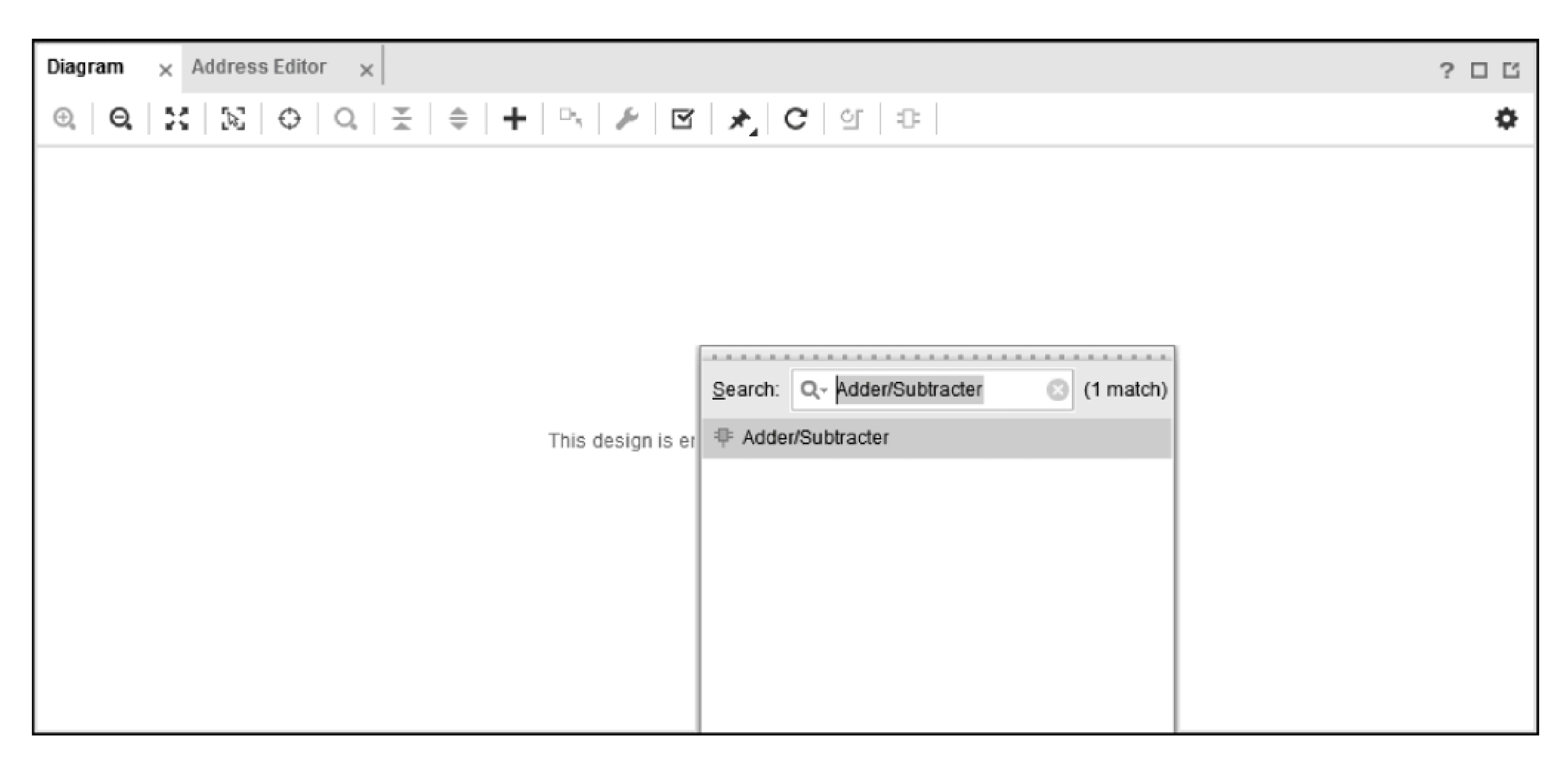


图 4-9　添加 IP 的界面

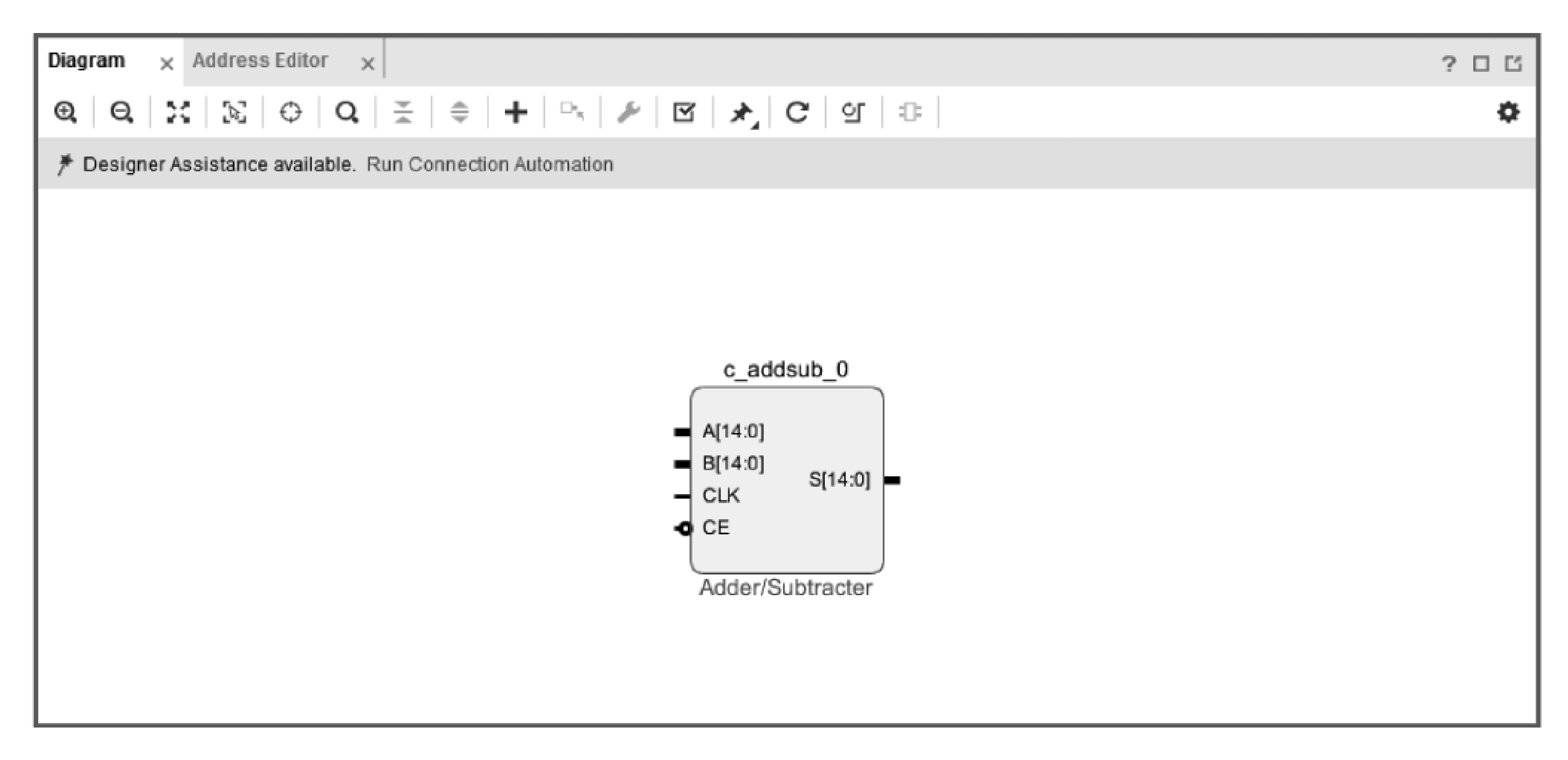


图 4-10　原理图开发界面

为了使该原理图可以和外部 I/O 相连接，需要将其引脚"引出"。如：将输入 A 引脚选中，右击，选择 Make External，可以看出图 4-12 中输入 A 引脚出现了输入的箭头图标，表示其可以接收外部的输入了。依次选中引脚，右击，选择 Make External 将其引出。如果想要将同一组输入连接到多个输入引脚，如 A、B 端口都连接到同一组输入 A_0[14:0]，可以先将 A 引出，然后将 B 连接到 A 端口，出现绿色对号的标志，表示允许连接。

原理图连接并将引脚引出完毕之后，可以在 Diagram 中右击，选择 Validate Design 或者在 Diagram 顶层菜单栏中单击 Validate Design 按钮或者按 F6 键，对原理图进行验证，确保连接无误。

为了使原理图方式设计的 IP 也能在使用 Verilog/VHDL 方式开发时被调用，可以将原理图封装成 HDL 的格式，后续可以通过调用该 HDL 封装文件调用到对应的原理图 IP。具体操作如下：右击，选择 Design Source 中需要 HDL 封装的原理图文件，如：design_1.bd，选择 Creat HDL Wrapper，弹出图 4-13 所示界面。其中 Copy generated wrapper to allow

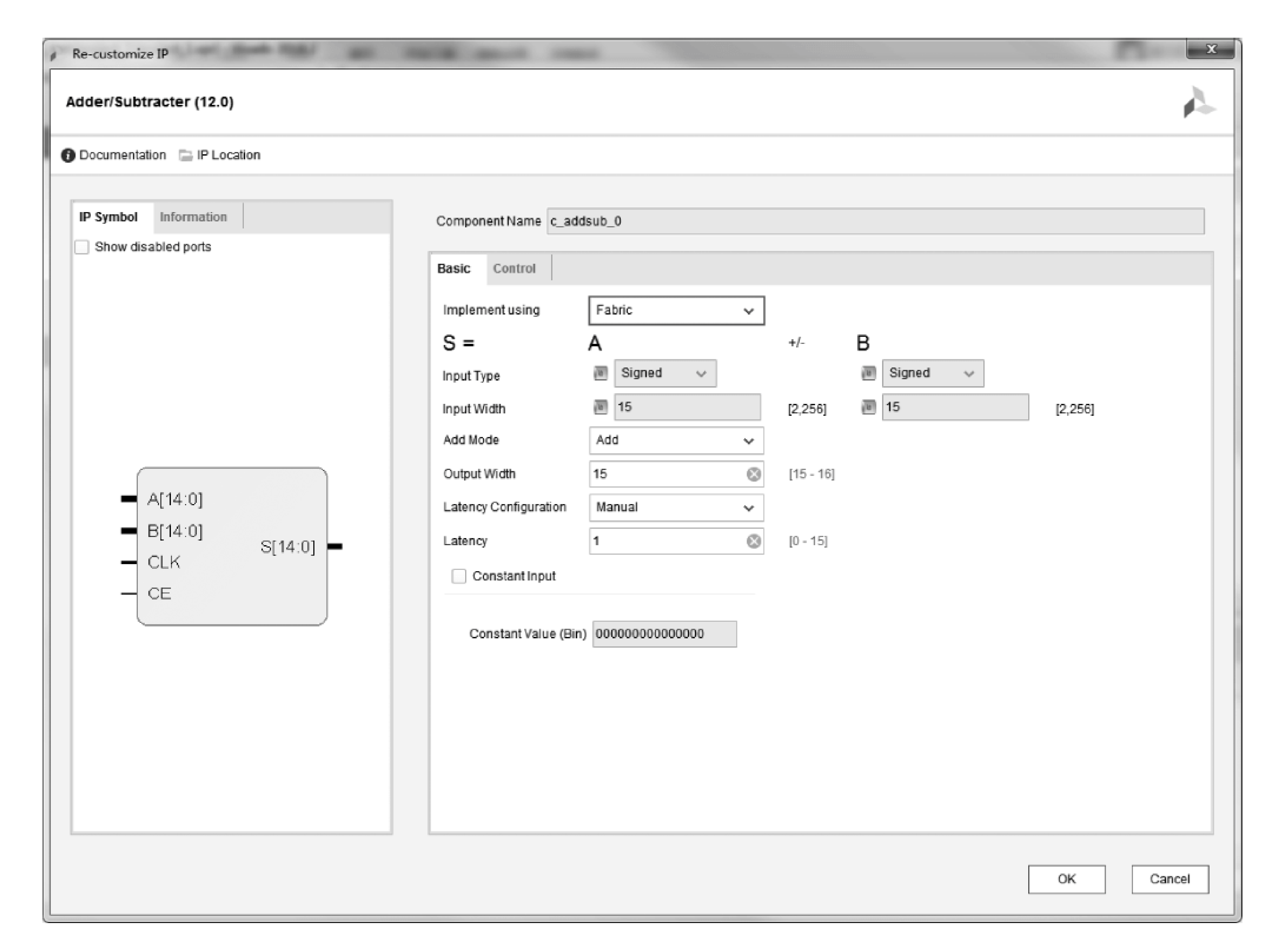


图 4-11　IP 配置界面

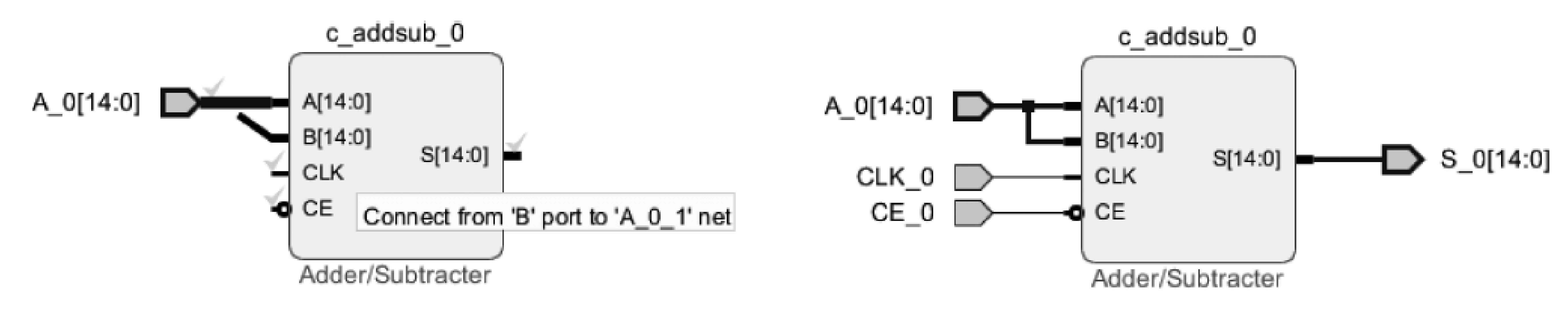


图 4-12　原理图引脚的“引出”

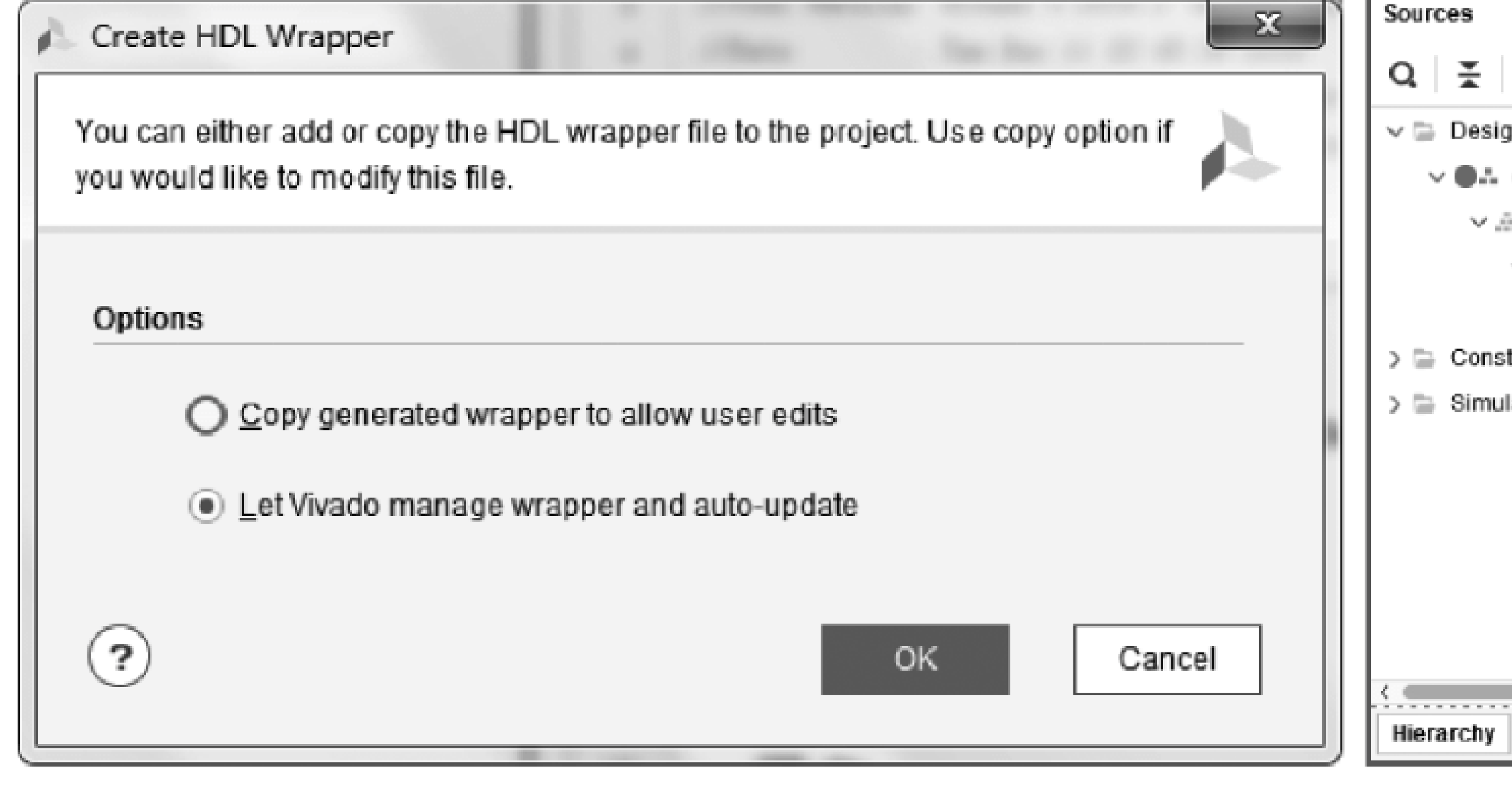




图 4-13　生成 HDL 封装

user edits 意思是使用者后续具有修改该 wrapper 的权利，而选择 Let Vivado manage wrapper and auto-update，意思是由 Vivado 来管理并且自动更新该 wrapper，而单纯地使用者将不能自行修改。

4.2.2　Verilog/VHDL 方式

Vivado 在支持原理图开发方式的同时，也支持硬件描述语言 Verilog/VHDL 的开发方式。这里将以简单的 LED 控制为例，说明如何使用硬件描述语言 Verilog/VHDL 进行开发。

在工程创建完成之后，右击，选择 Design Source 选择 Add Sources 添加或创建设计文件。如图 4-14 所示，选择 Create File，创建名为 led. v 的 Verilog HDL 源文件。VHDL 源文件的后缀名为. vhd。

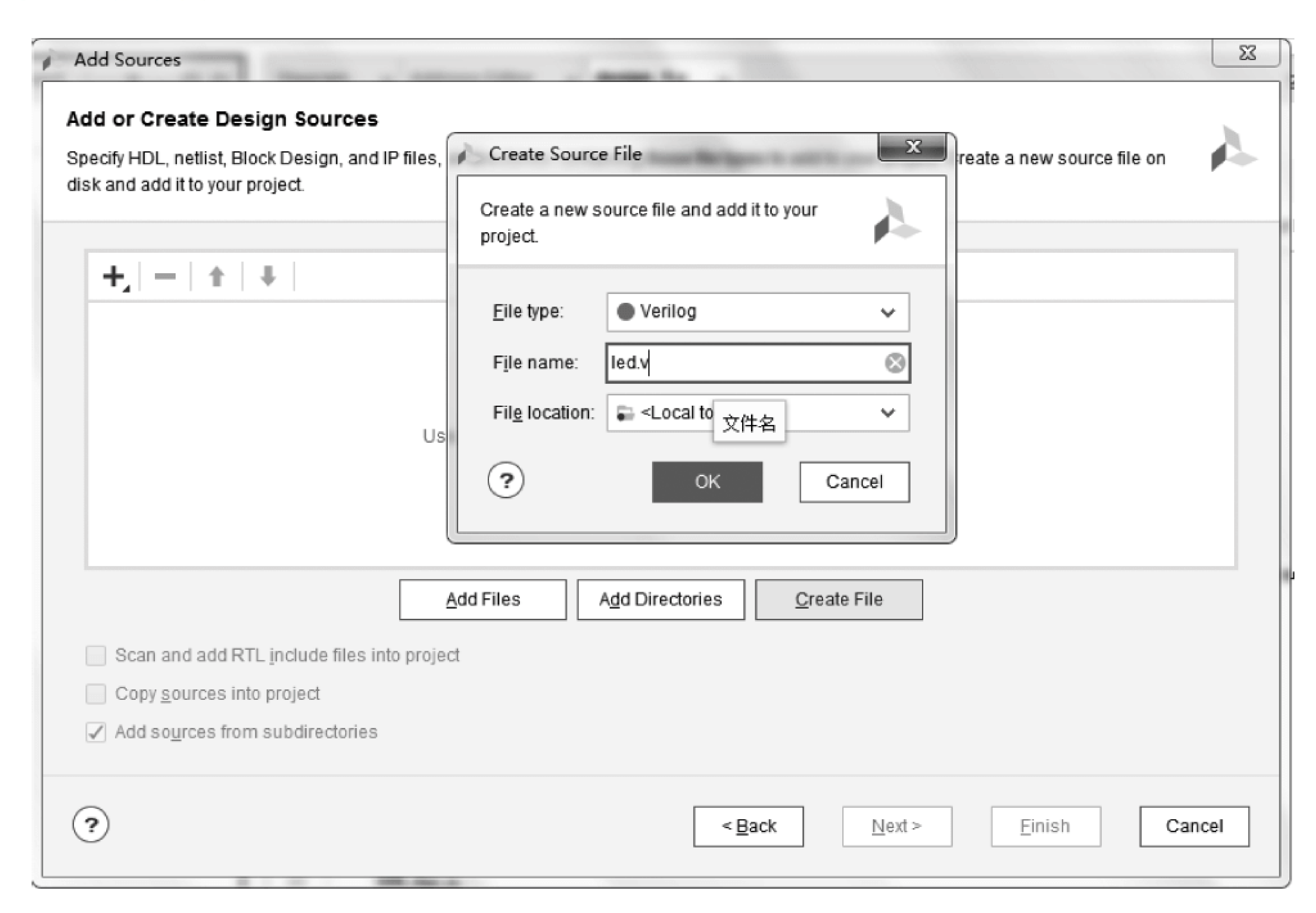


图 4-14　创建 HDL 文件

选择 Finish 后，Vivado 会弹出如图 4-15 所示的 Define Module 对话框，用户可在此定义模块名称和 I/O 引脚，也可以简单地跳过，后续在源文件中直接通过代码方式定义。

确定之后即可进入该 led. v 源文件的编辑界面，如图 4-16 所示。在此完成源代码的编写。

为了将该模块在其他设计中以原理图的方式被调用，或在被其他设计调用时不看到源代码，可以选择将上述 Verilog 编写的模块打包成 IP。通过选择 Create and Package New IP 打包 IP 核。在此之前需要将 led. v 设置为顶层文件。如果不需要，则可以再开发中直接使用. v 文件。

创建和打包 IP 时，可选择将整个项目、当前的 Block Design 或指定的目录进行打包（见图 4-17），还可创建 AXI4 的总线接口。

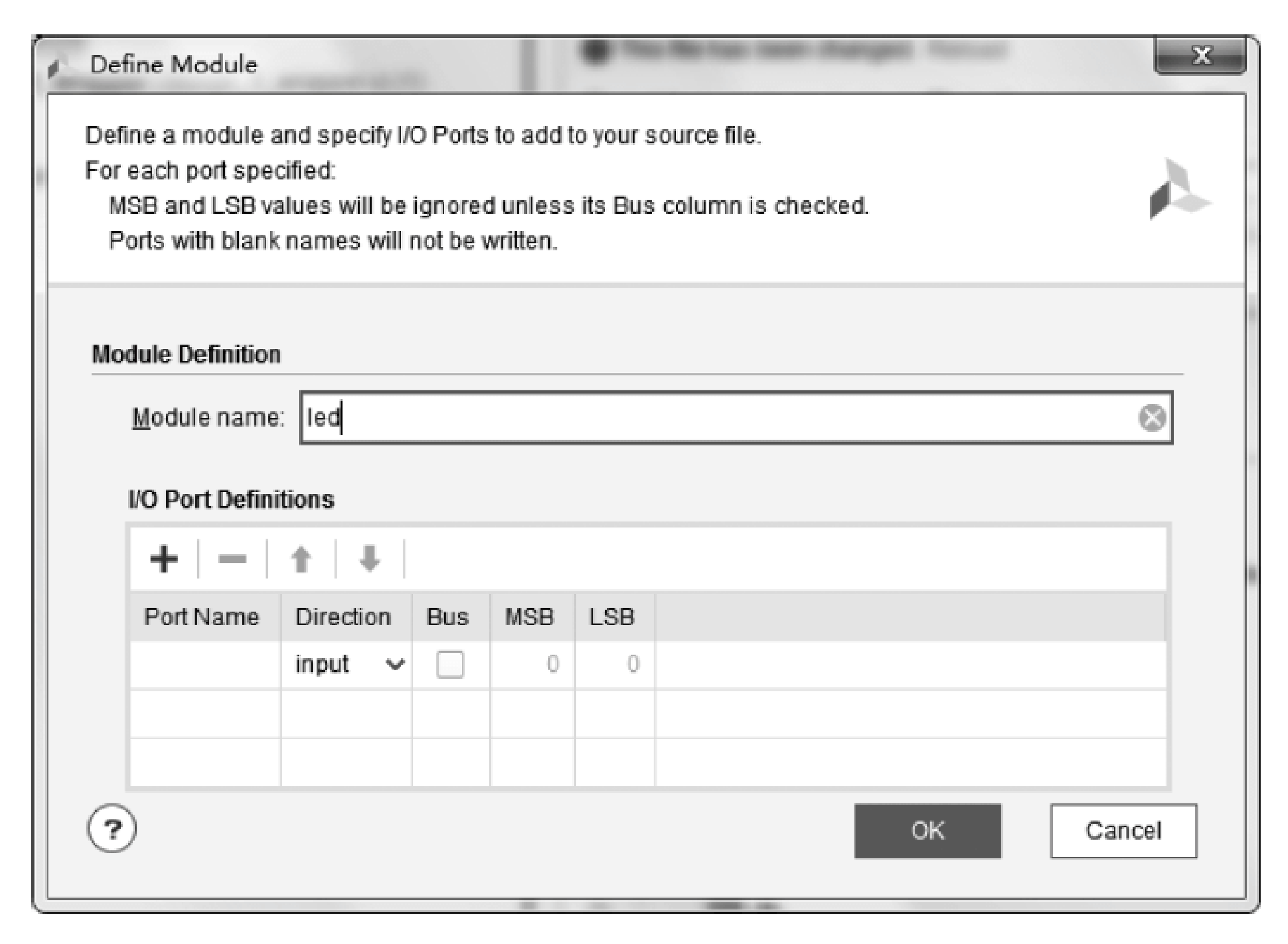


图 4-15 模块定义对话框

```
//Verilog 代码示例
module led(input [3:0] sw, output [3:0] led);
    assign led = sw;     //将 sw 端口直接与 led 相连
endmodule
```

图 4-16 Verilog 示例源代码

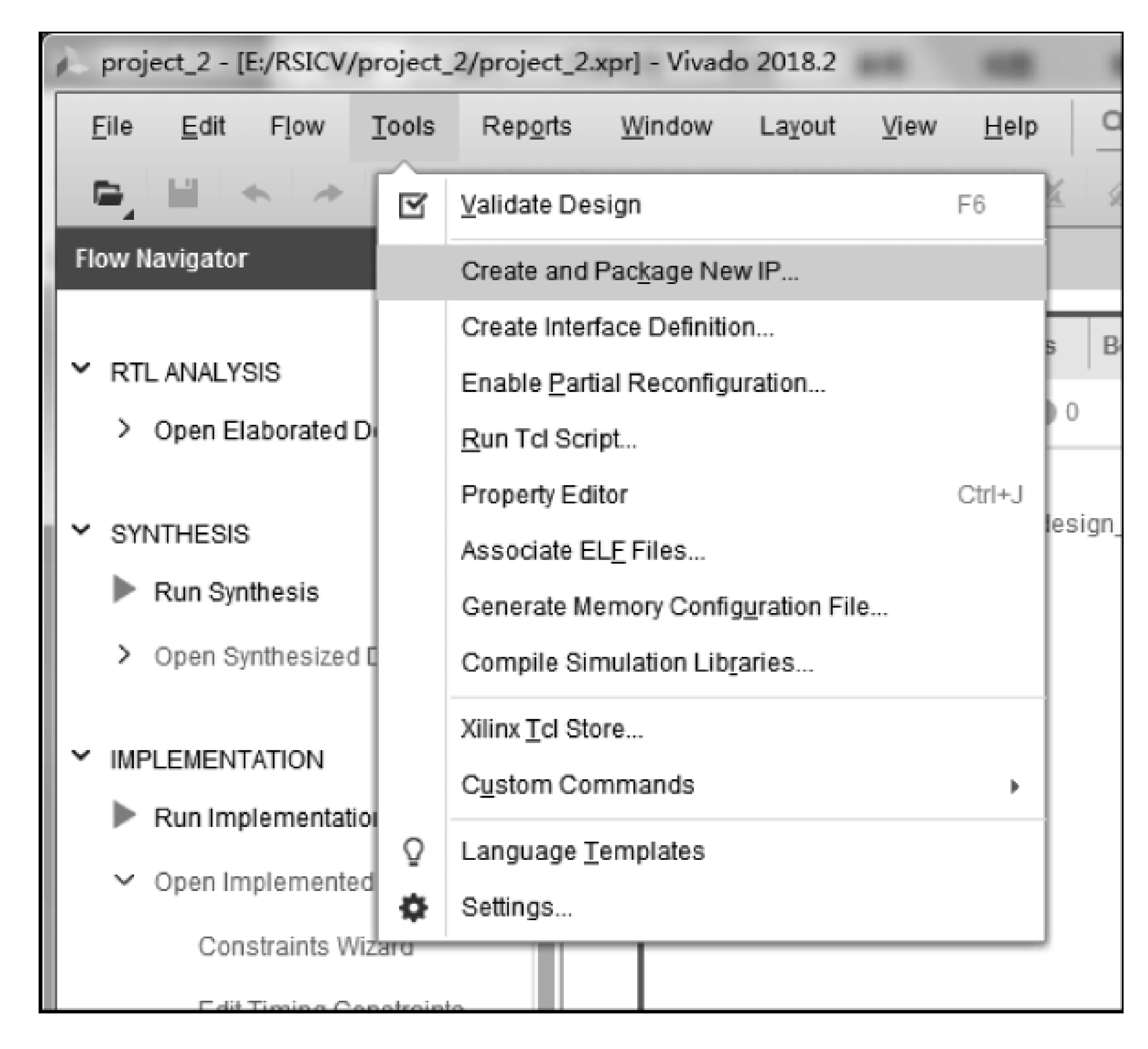


图 4-17 IP 打包

指定 IP 输出的位置，在图 4-18 选择“. xci”文件，这是定制 IP 输出的文件，该文件包含了定制 IP 核的所有信息，可以通过这个文件使用需要的 IP 核。

如果选择“Include IP generated files”则是包含所有 IP 生成的文件。单击 Next 按钮后，单击 Finish 按钮完成创建 IP 核。生成后需要设置打包 IP 核配置信息，最后选择 Review and Package，Package IP 完成 IP 的创建与打包。

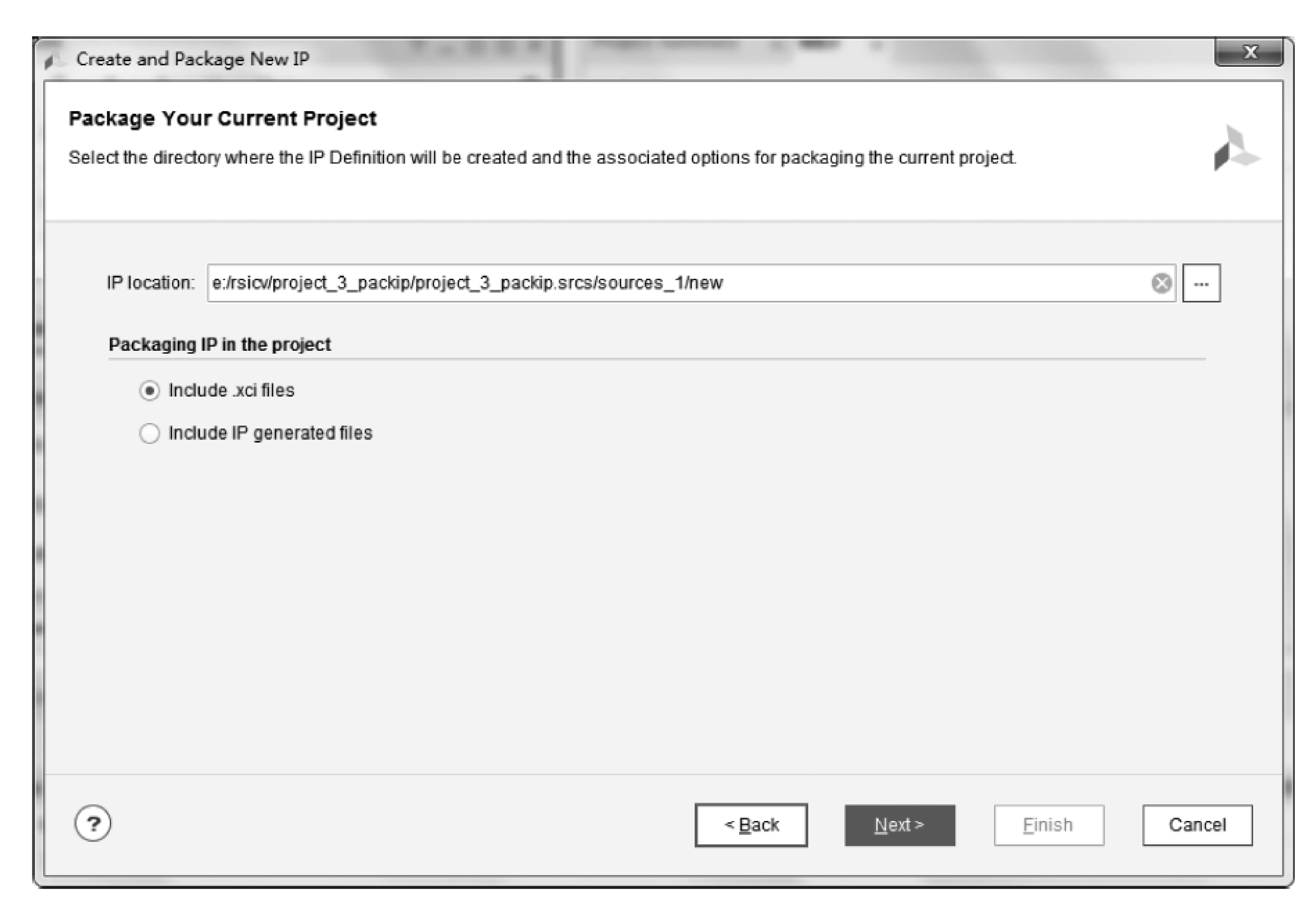


图 4-18　IP 打包时的选项

为了能够在使用时调用到上述产生的 IP，需要将 IP 核添加到库中。在 Block Design 中右击，选择 IP Setting，如图 4-19 所示选中 Repository，将上述 IP 添加进来。之后在使用时便可在 Block Design 中添加该用户定义的 IP 核，注意设置路径为 IP 核存放路径。

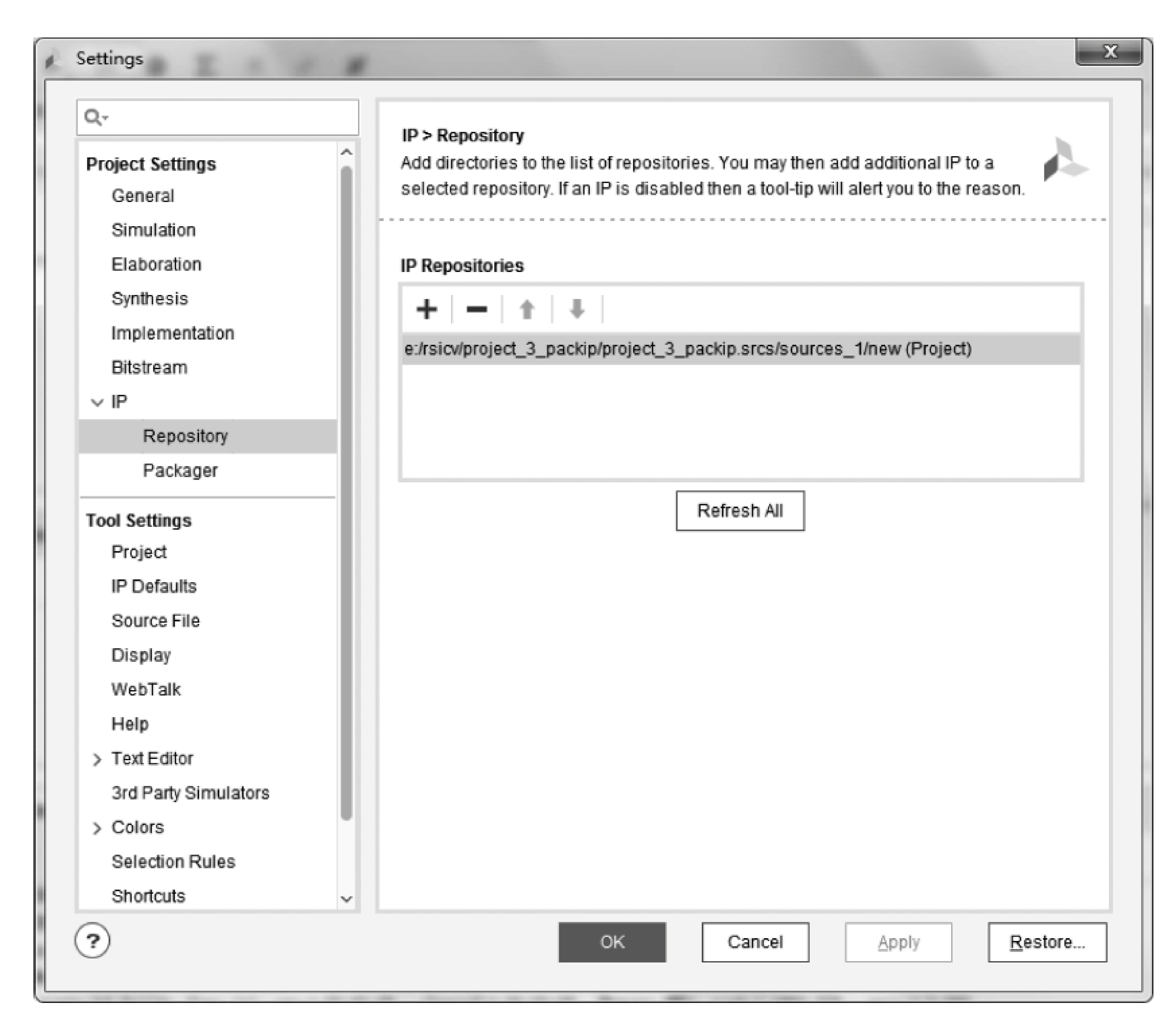


图 4-19　添加 IP Repository

4.3 仿真

上述介绍了如何通过不同的方式设计一个功能模块，在完成设计之后需要快速验证该模块是否正确。为了及时快捷地验证所设计的模块的正确性，通常在下板之前先要对系统进行仿真。尤其是功能仿真，是组成原理实验中经常用到的手段。

在进行仿真时，通常的做法是设计一个测试文件，通过该文件产生模块所需要的激励输入信号，并观察模块的输出。该测试文件就叫 Testbench。

本节以验证 4.2.2 节中 led 模块为例，介绍 Testbench 如何创建、编写以及如何通过加载 Testbench 对模块进行仿真。

1. 创建 Testbench

在工程主界面的 Design Source 中添加 Testbench，在 Source 中右击，选择 Add Source，然后选择如图 4-20 所示的 Add or create simulation sources。选择 Create File 创建 Testbench 文件，即可进入 Testbench 编辑界面。

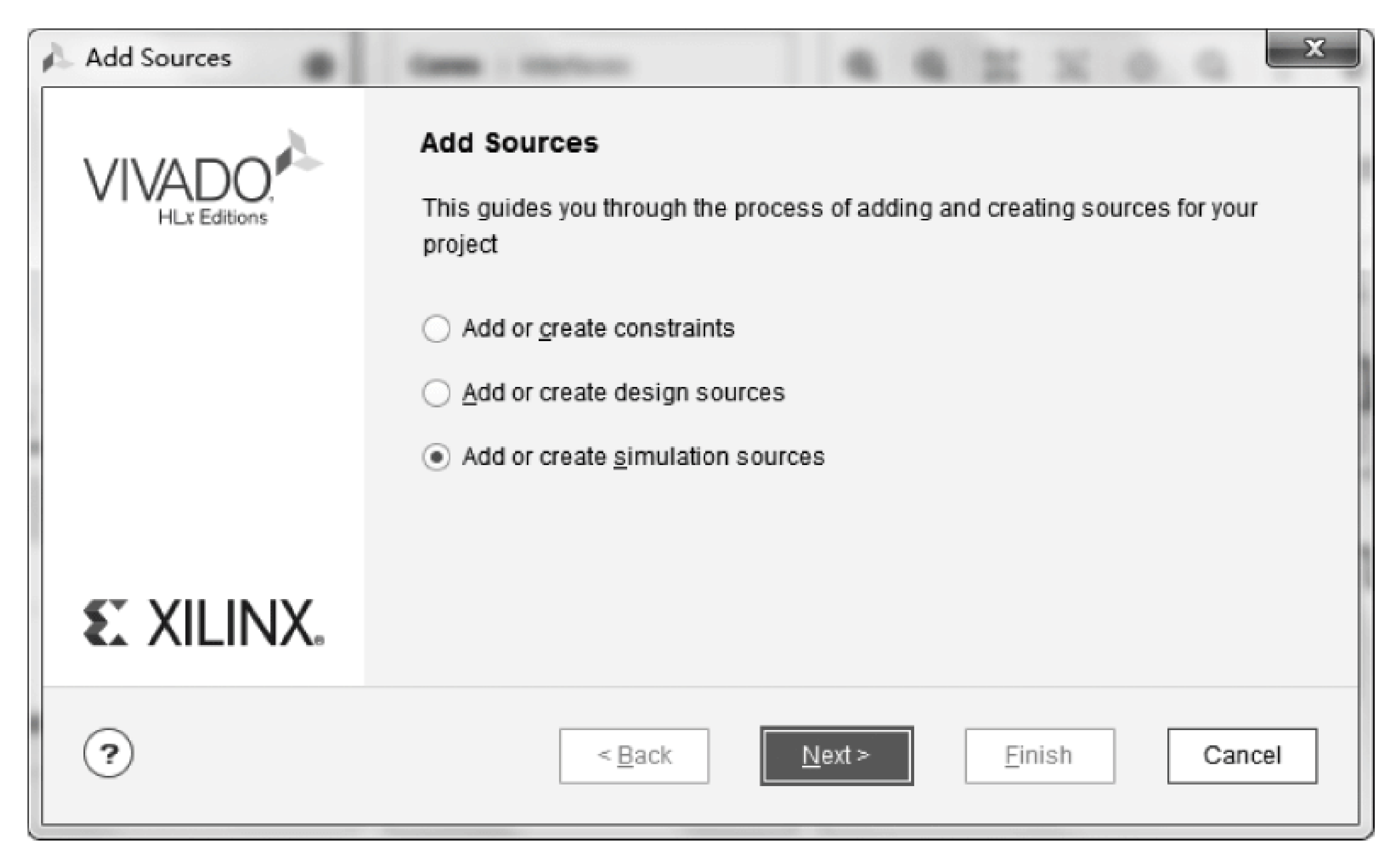


图 4-20　建立 Testbench 文件

2. 编写 Testbench

编写 Testbench 的目的就是为待仿真的模块提供输入并获取输出。在 Testbench 中可以为输入变量设定不同的值，而且可以为其设定时间。

举例说明：用来仿真 led 模块的 Testbench 示例代码如下：

```
module sim(                        //Testbench 模块名，为顶层模块且没有输入/输出
    );
    reg [3:0] sw;                  //Testbench 内部变量
    wire [3:0] led;                //Testbench 内部变量
    initial begin
        sw = 4'b0000;              //在仿真时刻 0，为 sw 指定输入值：4 位的二进制 0000
    end
    initial begin
```

```
        #20                            //延时 20 个时间单位
        sw = 4'b0001;                  //在仿真时刻 20 时,修改 sw 的输入值为 0001
        #20                            //延时 20 个时间单位
        sw = 4'b0010;                  //在仿真时刻 40 时,修改 sw 的输入值为 0010
        #20                            //延时 20 个时间单位
        sw = 4'b0100;                  //在仿真时刻 60 时,修改 sw 的输入值为 0100
        #20                            //延时 20 个时间单位
        sw = 4'b1000;                  //在仿真时刻 80 时,修改 sw 的输入值为 1000
        end
        led led_1(.sw(sw), .led(led)); //将 Testbench 的内部变量与待验证的 led 模块的
                                       //输入/输出相连
    Endmodule
```

3. 运行仿真

Testbench 编写完成后,在左侧工程管理栏中选择 Simulation→Run Simulation 可进行仿真验证。

Run Simulation 有五种仿真模式,可根据需要进行选择。

(1) Run behavioral simulation:行为级仿真,通常也称为前仿真或功能仿真,是对设计输入的直接仿真。

(2) Post-synthesis function simulation:综合后的功能仿真。

(3) Post-synthesis timing simulation:综合后带时序信息的仿真。

(4) Post-implementation function simulation:针对具体 FPGA 芯片实现之后的功能仿真。

(5) Post-implementation timing simulation:针对具体 FPGA 芯片实现之后的时序仿真,该仿真最接近真实的时序波形。

在组成原理实验中,通常综合前或综合后的功能仿真即可。在此运行行为仿真(run behavioral simulation)后可查看波形,如图 4-21 所示。可以通过查看波形根据输入/输出的变化验证模块的功能。

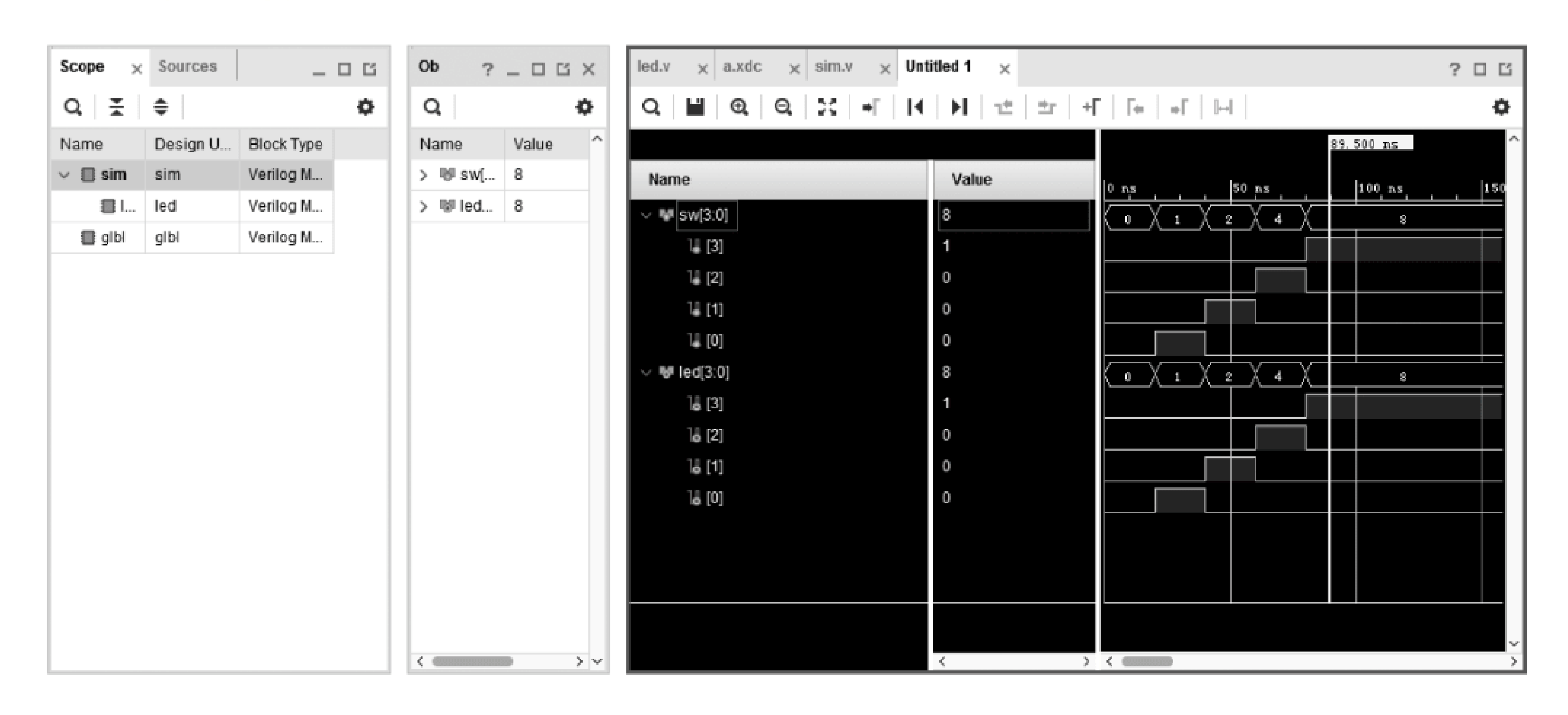


图 4-21　led 模块的功能仿真结果

Scope 面板可选择要仿真的模块，选择后仿真的输入/输出便会出现在 Objects 面板中，如图 4-22 所示。

在 Objects 面板中，可选择 Add to Wave Window 添加需要观察的信号到仿真窗口，如图 4-23 所示。仿真波形的观察可以通过窗口的推拉进行缩放。

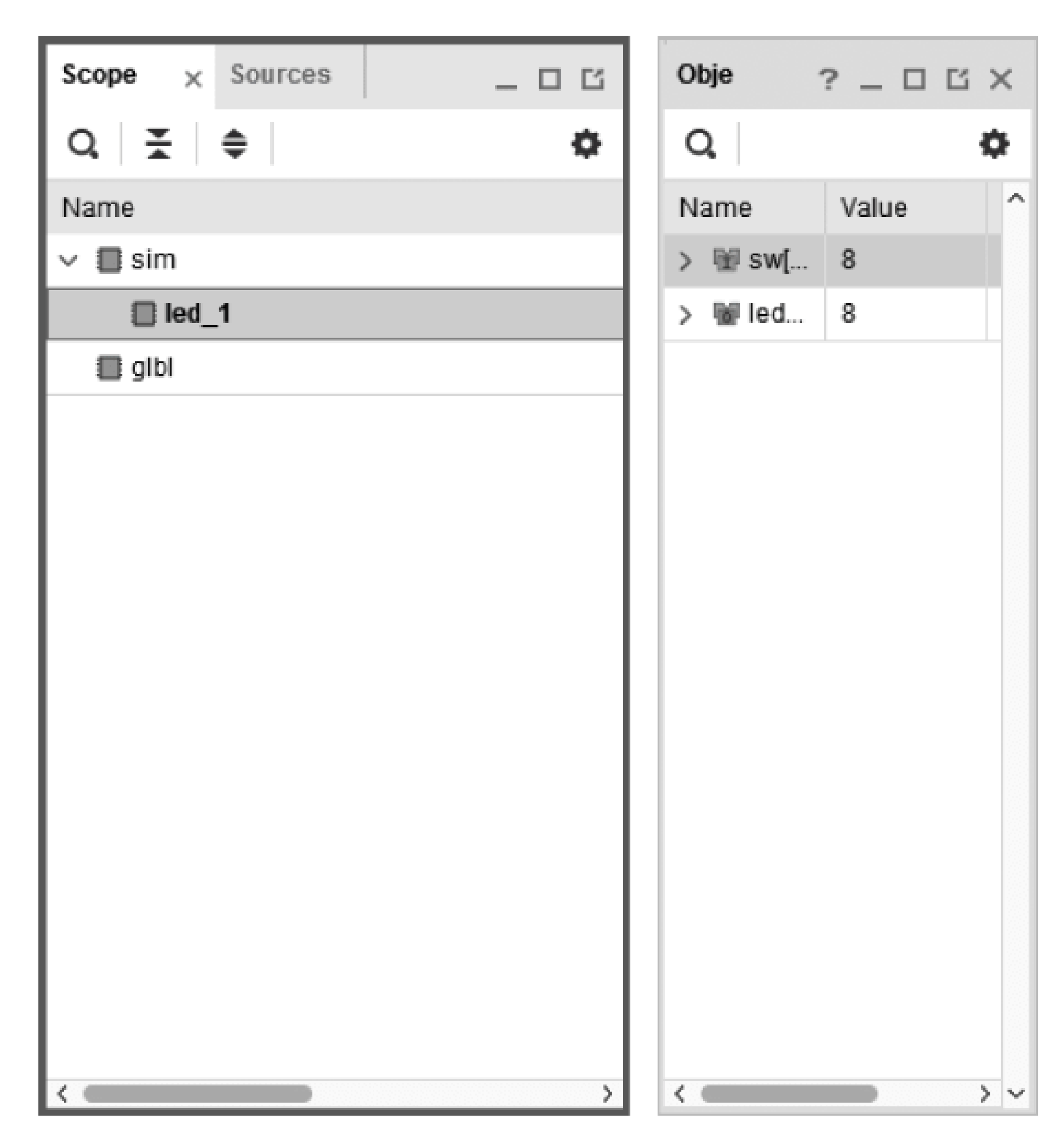


图 4-22　仿真模块的选择

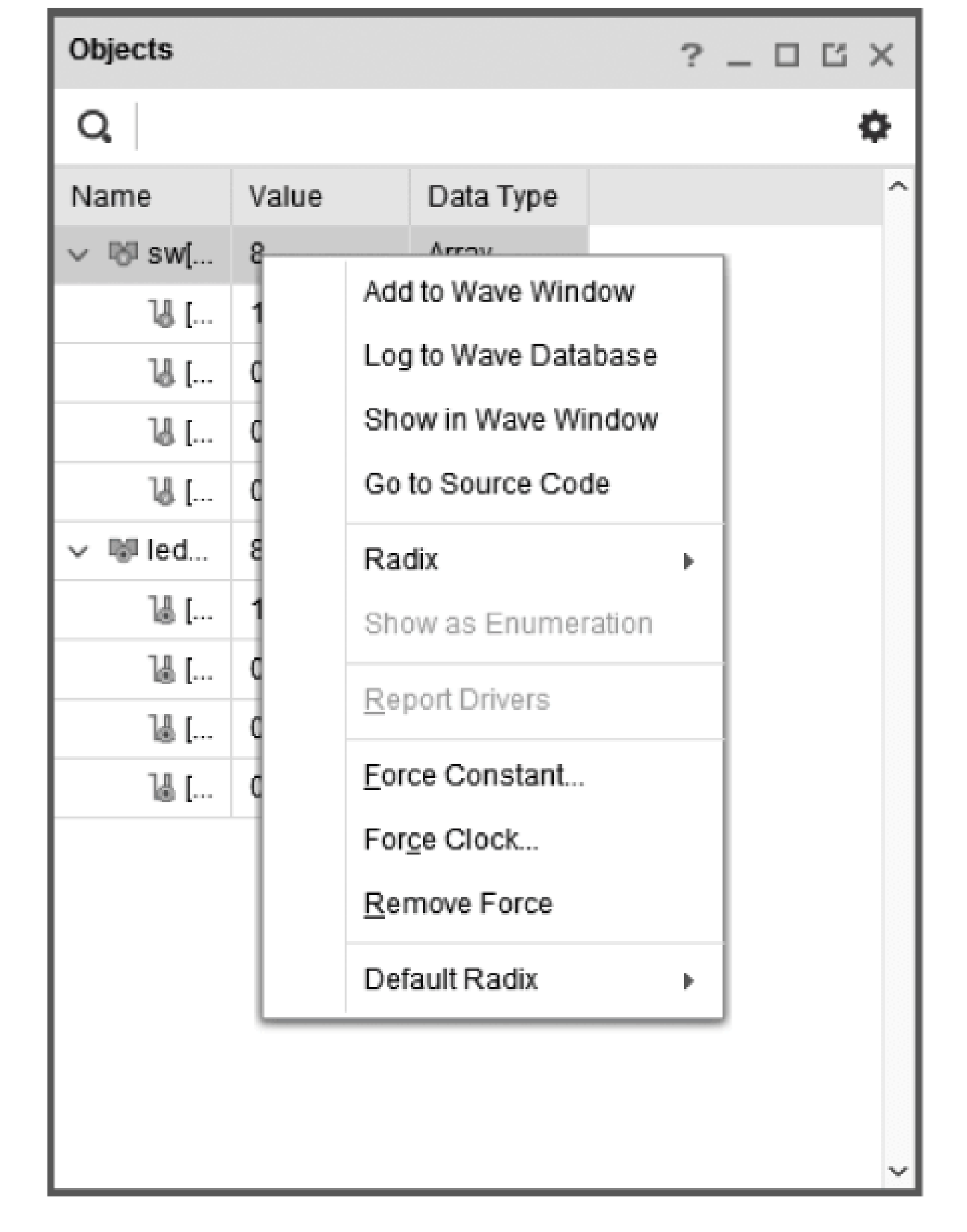


图 4-23　观察信号波形的添加

仿真时，可根据需要进行 Restart、Run All(一直运行)、Run for 10us(限时运行)、Step、Relaunch Simulation 等操作控制仿真的时间。

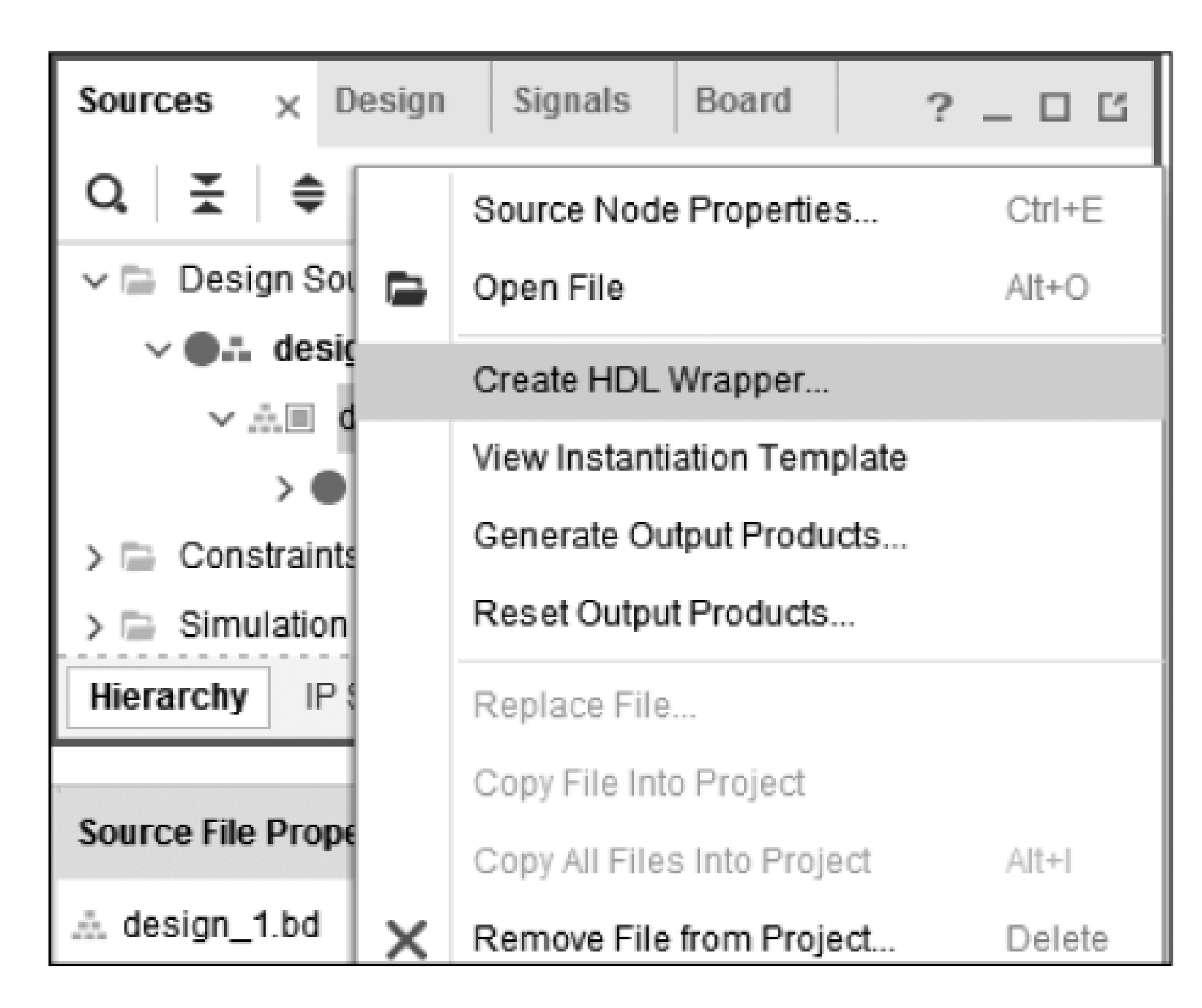
4.4　综合

不管是原理图方式还是 Verilog/VHDL 方式的设计输入，都需要经过综合后才能进行后续的布局布线以及板级实现等。所谓综合，就是将电路原理图或者硬件语言描述文件装换成综合网表的过程。综合网表中除了包含与门、非门等组合逻辑和寄存器等时序逻辑之外，还包含 FPGA 特有的各种原语(Primitive)，例如 LUT、BRAM、DSP48，甚至 PowerPC、PCIe 等硬核模块，以及这些模块的属性和约束信息。

选定一个模块进行综合之前，需要将其设置为顶层模块。如果使用原理图方式进行设计，还需要对其进行打包，再进行综合，如图 4-24 所示。然后选择 Flow Navigator 中的 Run Synthesis 进行综合。

如果综合出现错误，则需要根据提示进行相应的设计修改。综合成功后可根据提示查看综合设计以及综合报告。上述 led 模块综合后的原理图，如图 4-25 所示。

综合后可在 Reports 面板中打开 report_utilization 查看详细的资源利用率等信息，以及在 synthesis_report 中查看综合过程的相关信息。在较为复杂的设计中，通常需要借助综合的报告调整自己的设计。当然在工程管理栏的 Synthesis 中可以查看更详细的综合报告。



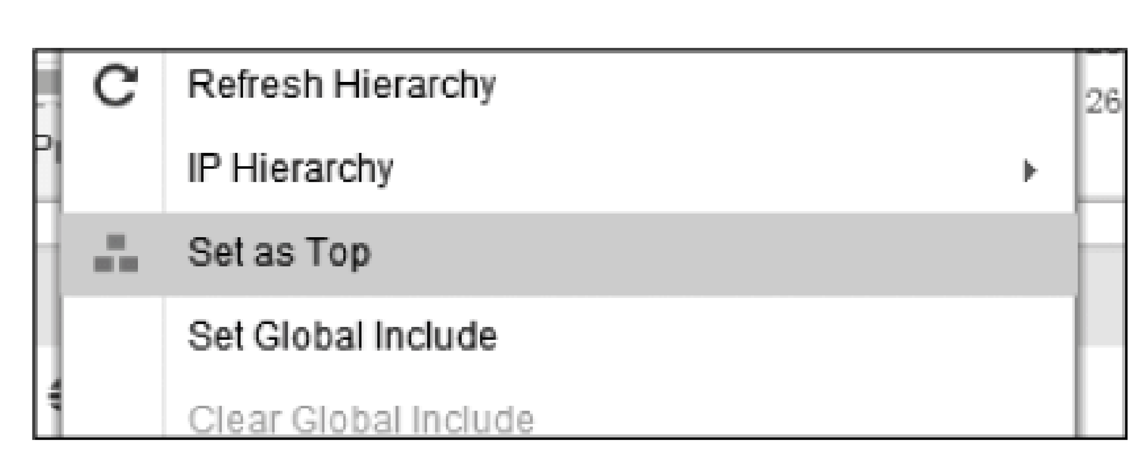


图 4-24　综合前准备

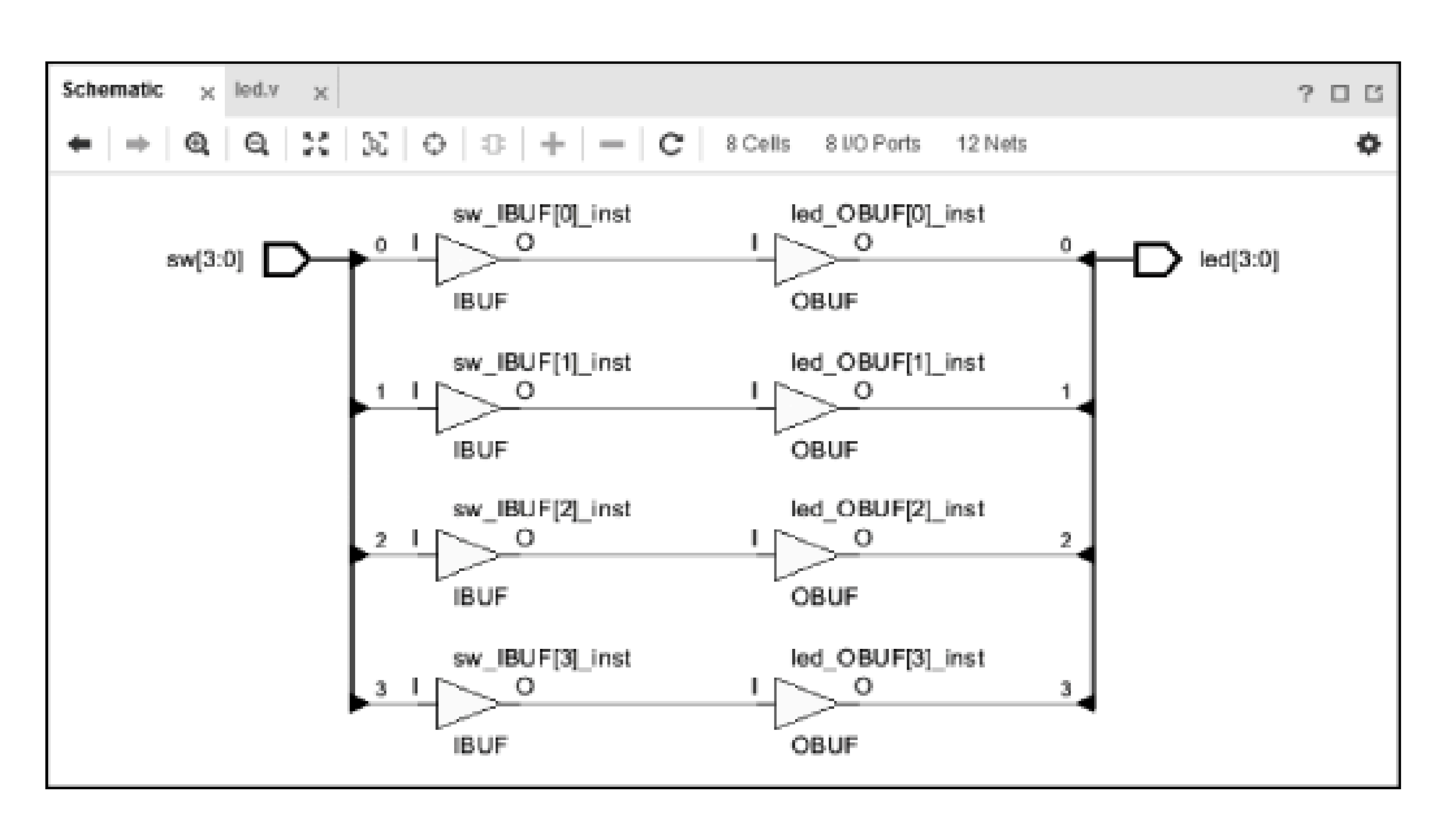


图 4-25　led 模块综合后的原理图

4.5　引脚绑定(I/O 处理)

经过上面的输入设计及综合之后,用户完成了一个功能模块的设计。虽然该功能模块在仿真时可以借助 Testbench 给它加载输入信号并通过 Testbench 读取它的输出信号,但当把它下载到 FPGA 芯片时,它还是一个没有与外部连接起来的孤立的模块。因此,在下载之前,先要为其进行“引脚绑定”操作。如图 4-26 所示,“引脚绑定”的意思就是将用户设计的功能模块的逻辑输入/输出和 FPGA 芯片的物理 I/O 引脚(如 PL 端与 LD0～LD3 连接的引脚)或监控模块(如与 PYNQ 的 PS 端连接的通道)的输入/输出连接起来,以便通过 FPGA 的物理引脚连接的外设或监控模块(可以通过 Python 程序)对用户的模块进行输入/输出,从而完成用户模块的运行及测试验证。

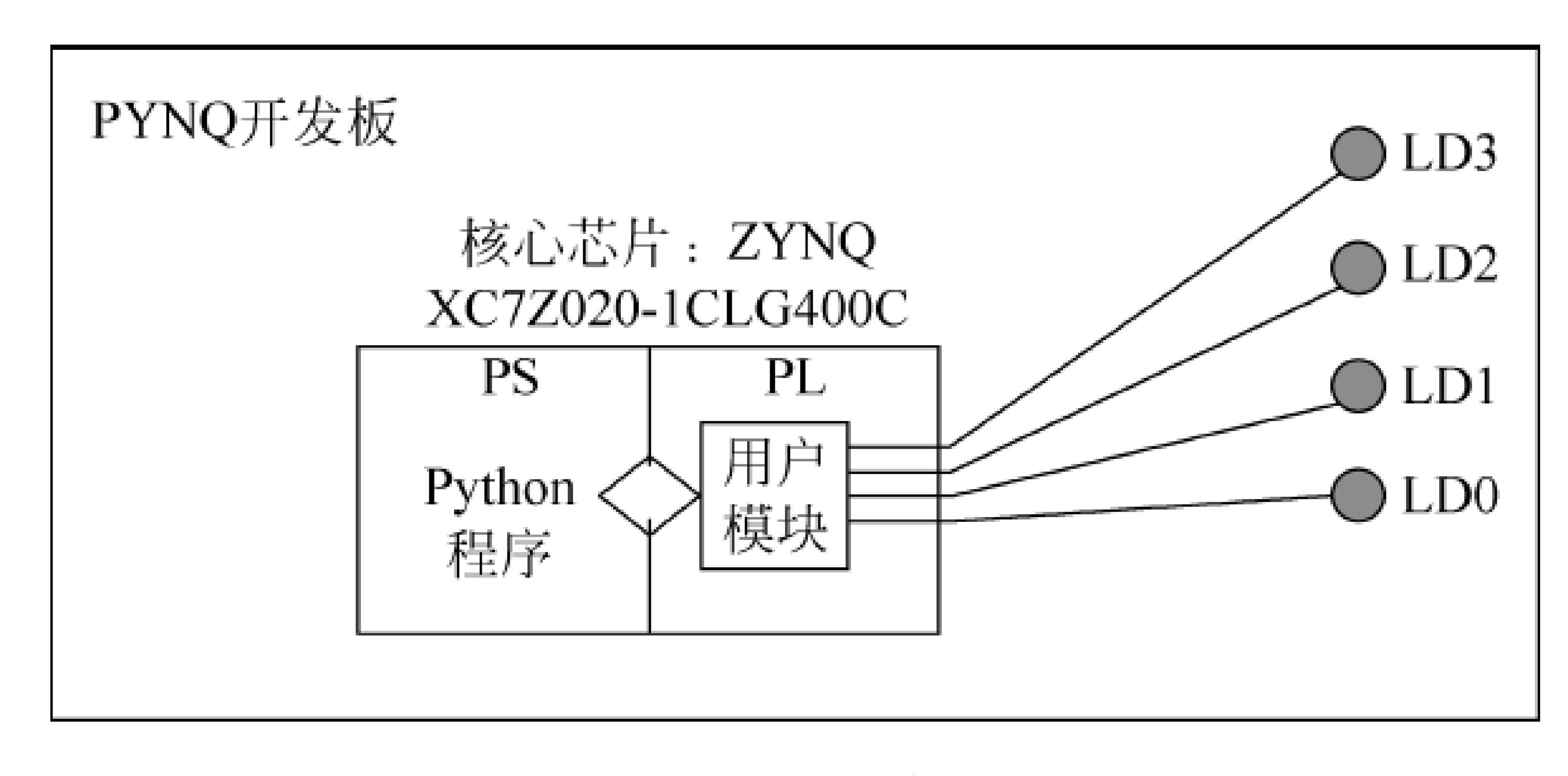


图 4-26　引脚绑定示意图

4.5.1 物理引脚的绑定

以 PYNQ 开发板为例，当需要把用户模块的 I/O 与 PL 部分的物理引脚进行绑定时，Vivado 支持直接编写约束文件.xdc 进行引脚约束。可在 Design Source 中的 Constraints 中新建引脚约束文件.xdc 并根据需要进行编写。同样支持综合后在 I/O Ports 面板中对 I/O 进行绑定。在工程综合完成后，选择左侧 SYNTHESIS 中的 Open Synthesized Design，再选择右上角下拉菜单，单击 I/O Planning 进行引脚绑定，操作如图 4-27 所示。

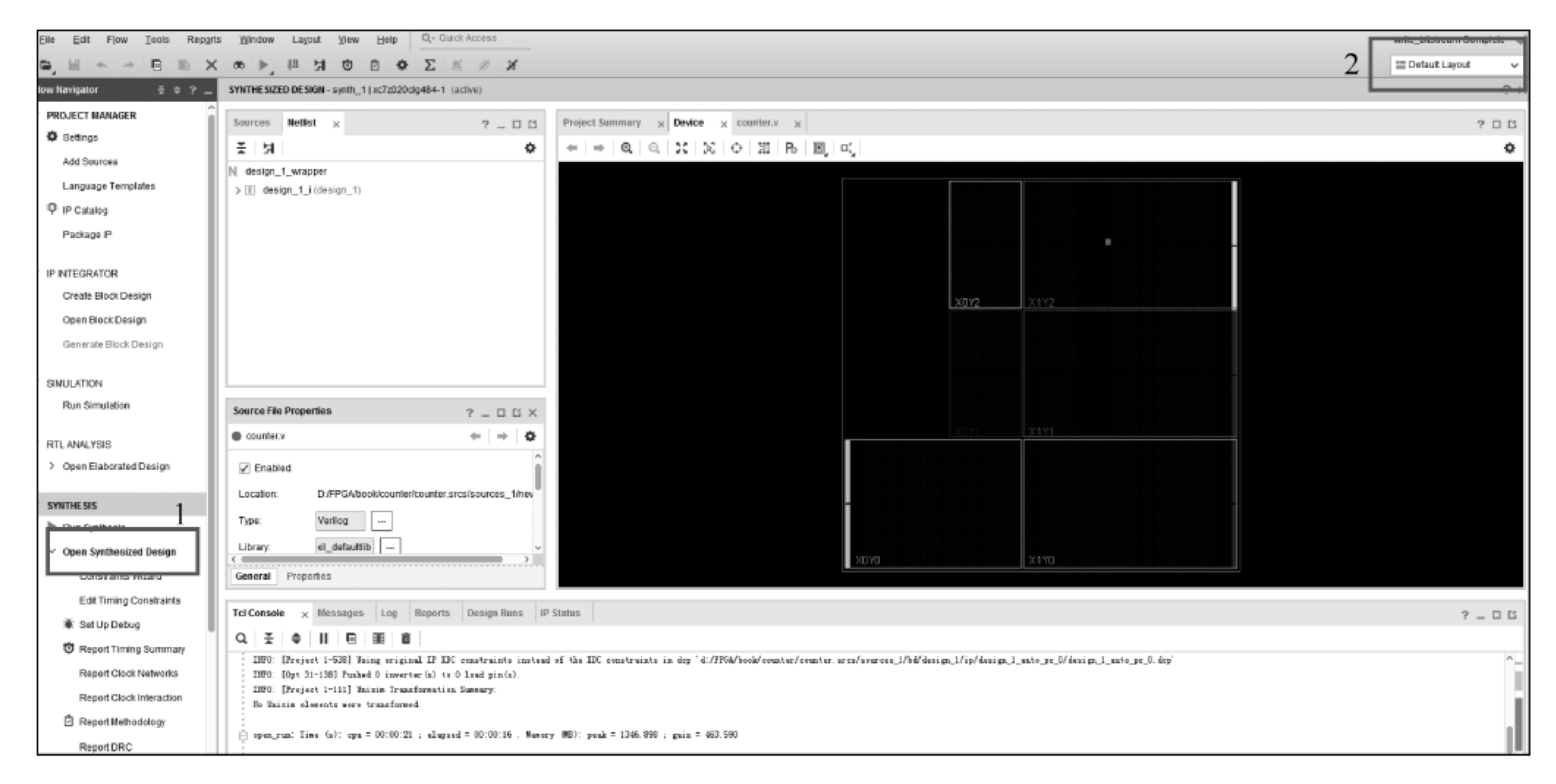


图 4-27 综合后的设计

也可以通过如图 4-28 所示界面选择 Vivado 菜单栏 Layout→I/O Planning 对引脚进行绑定。

如图 4-29 所示，在 I/O Ports 面板中可对约束引脚、IO 电平标准以及其他进行相关配置。引脚设置需要参考 PYNQ 原理图将对应引脚进行连接。PYNQ 原理图和其他 PYNQ 相关文档可在 http://www.tul.com.tw/ProductsPYNQ-Z2.html 中下载。

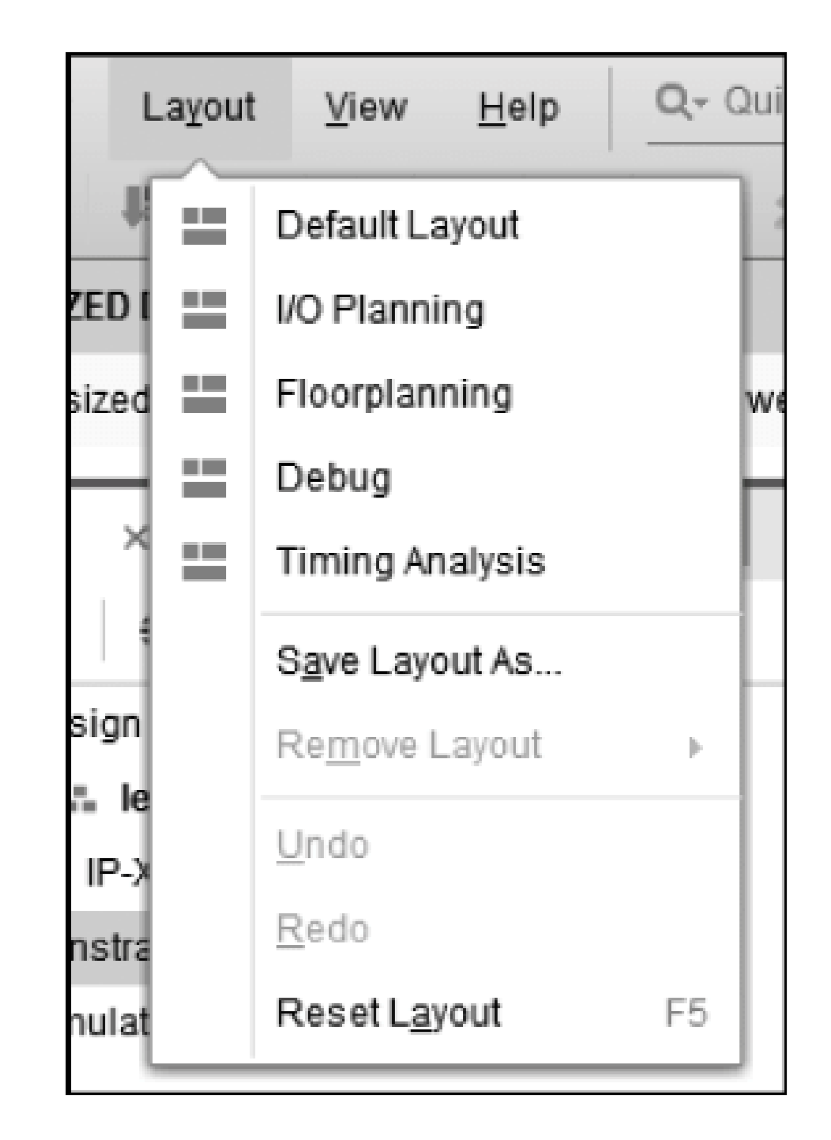


图 4-28 选择输入/输出布局

4.5.2 与监控模块(PS)的连接

除了与 PL 的引脚直接相连进行输入/输出外，在基于 PYNQ 的开发中，也可以通过将功能模块与 PS 端相连，从而通过 PS 端运行的 Python 软件完成 PL 上功能模块的输入/输出。

1. PS/PL 连接通道介绍

我们知道，在基于 PYNQ 的组成原理实验中，用户的硬件模块是在 PL 中运行的。如果想把 PS 端作为监控模块并在 PS 上运行 Python 与 PL 端的用户模块进行交互，就需要将 PL 中的用户模块的 I/O 与 PS 端连接起来。

PYNQ 的核心芯片 Zynq SoC 芯片中 PL 与 PS 之间的连接有如下几种物理通道，如图 4-30 所示。

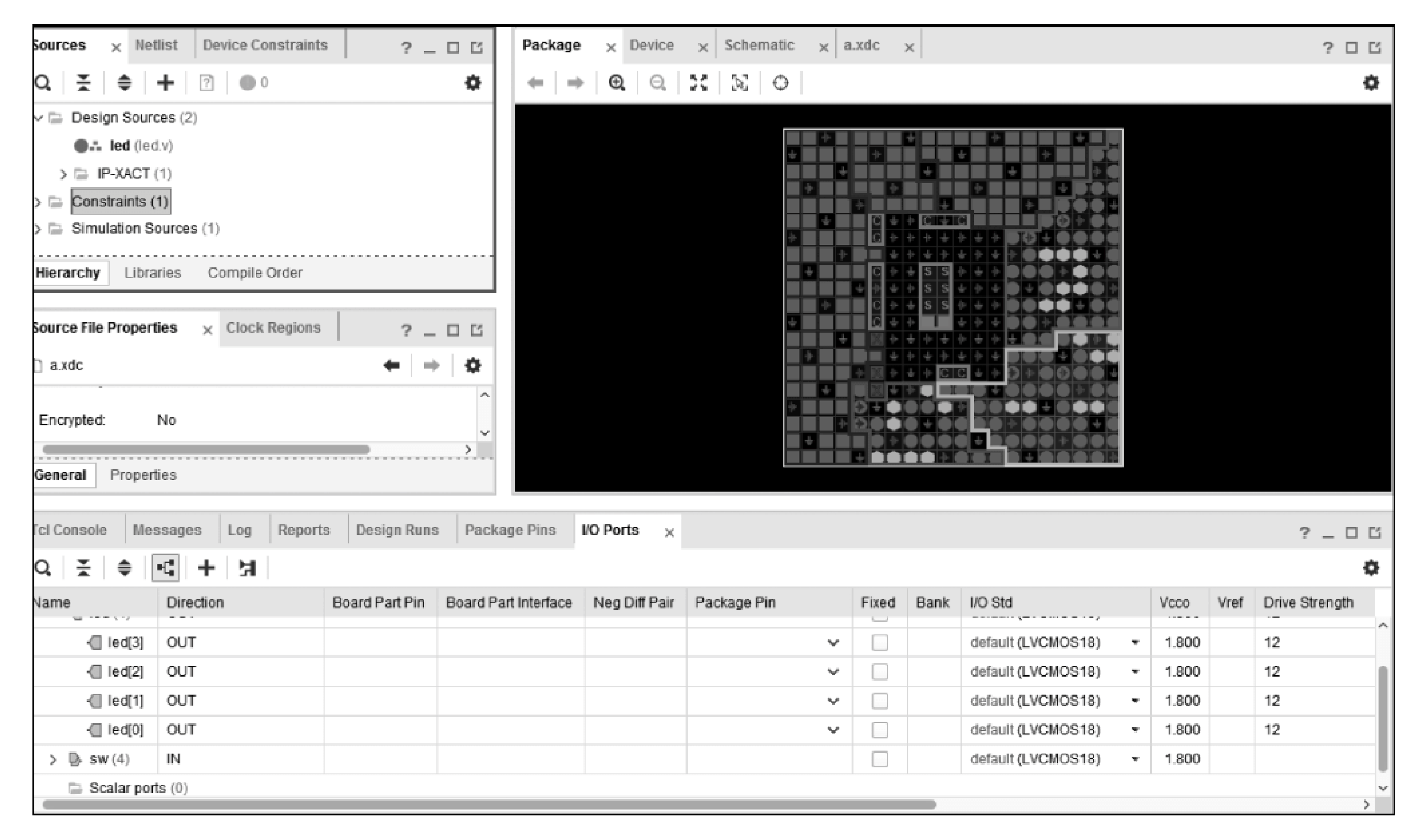


图 4-29　I/O Ports 的布局

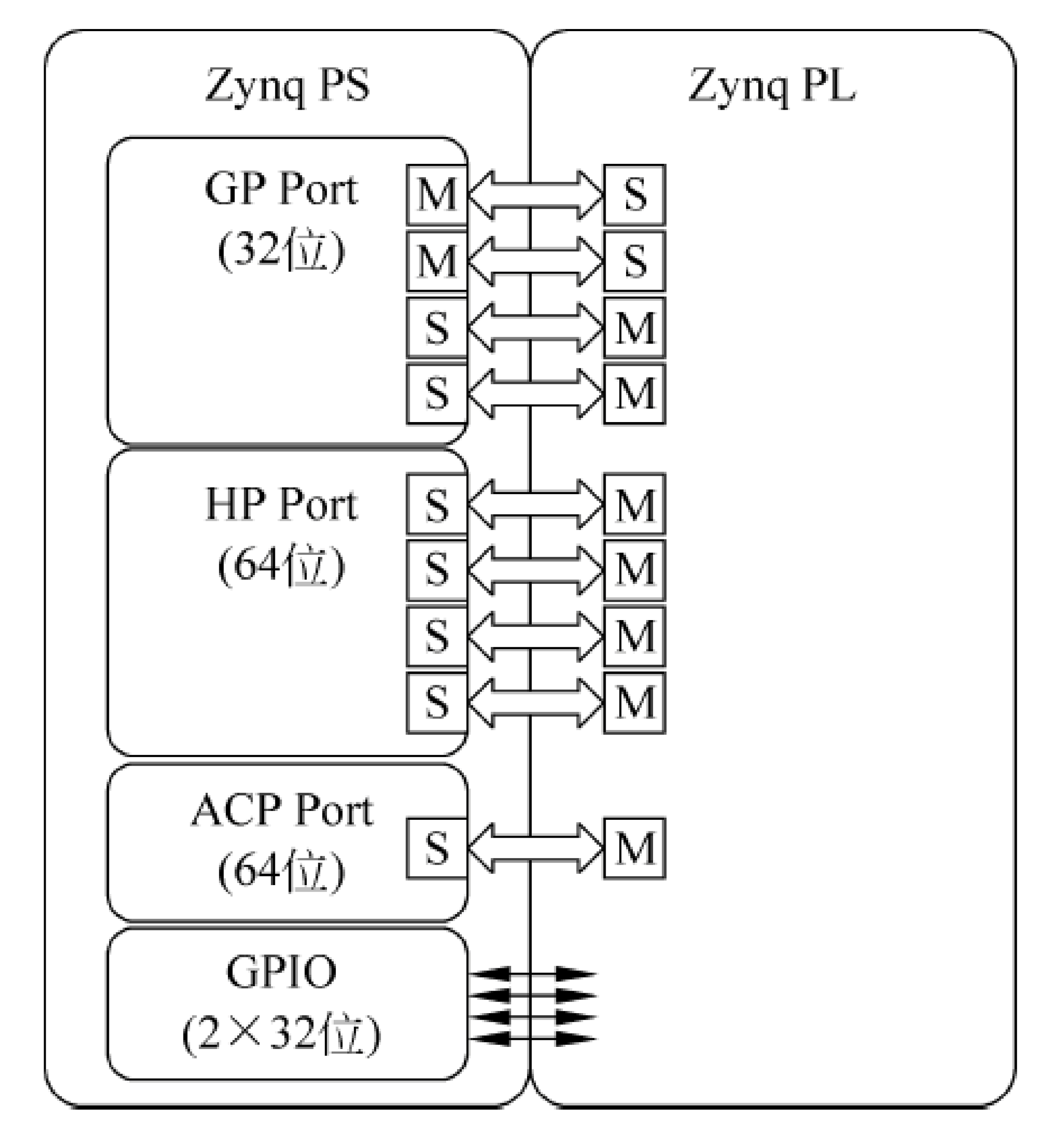


图 4-30　PS 和 PL 的物理连接通道

图中的 Zynq 在 PS 和 PL 之间有若干个 AXI 接口。Zynq SoC 芯片的 PL 与 PS 之间有 4 个 32 位的 GP(General Purpose)通道、4 个 64 位的 HP(High Performance)通道、1 个 64 位的 ACP(Accelerator Coherency Port)通道以及 2 个 32 位的 GPIO 通道。PS 和 PL 可以分别以主(M，Master)或从(S，Slave)的方式来使用上述通道。从设备只能被主设备控制，而不能对主设备控制。物理连接通道内的功能如下：

(1) GP Port 最大读写数据宽度为 32 位。不支持 AXI 的突发模式，比较适合少量的数据传输。

(2) HP Port 最大读写数据宽度为 64 位。支持 AXI 的突发模式，适合大量的数据传输。可访问 PS 端的 DDR 和 OCM(On-Chip Memory)。

(3) ACP Port 可直接访问 PS 端的 Cache。通过该接口 PL 端的逻辑可以直接访问 PS 端的 Cache，而且 Cache 一致性由 SCU 模块保证。因此 PL 逻辑可以直接从 Cache 中拿到 CPU 的计算结果，同时也可以第一时间将逻辑加速结果送至 CPU Cache 中，因此其延时很小，适合作专用指令加速模块接口。

(4) GPIO 是一个非常简单的接口，不需要复杂的地址映射。从 PS 到 PL 有多达 64 个 GPIO(EMIO)，PS 与 PL 可直接进行交互，使用简单方便。可用作一些控制信号如复位、中断等，PL 中不需要 IP 就可以使用它们。

其他几种可以用于管理 Zynq PS 和 PL 接口间数据移动的方式：

(1) MMIO(Memory Mapped IO)：任何连接到 AXI GP 从端口的 IP 都将映射到系统内存映射中，可以使用 MMIO 读取/写入内存映射位置。MMIO 读或写命令是单次传输 32 位数据到内存位置或从内存位置读出数据。由于不支持突发指令，MMIO 最适合在 IP 连接到 AXI GP 从端口之间读写少量数据。MMIO 允许 Python 对象访问映射到系统内存中的地址，特别是可以访问 PL 中外设的寄存器和地址空间。

(2) Xlnk(Memory allocation) Xlnk 用于分配连续内存。连接到 AXI HP 主端口或 ACP 主端口的 IP 可以访问 PS DRAM。在 PL 中的 IP 访问 DRAM 之前，必须先分配(保留)一些内存供 IP 使用，以及传递给 IP 的内存的大小和地址。Python 或 Numpy 中的数组将分配到虚拟内存中的某个位置。分配内存的物理内存地址必须在 PL 中提供给 IP。Xlnk 可以分配内存，也可以提供物理指针，它还可以分配连续内存以便 PL IP 更有效使用。Xlnk 可以使用 Python NumPy 包分配数组。允许使用 NumPy 指定数组的数据类型和大小/形状。DMA 也隐式地使用 Xlnk 来分配内存。

(3) DMA(Direct Memory Access) AXI 接口通常用于高性能流应用程序。AXI 可以通过 DMA 与 Zynq AXI HP 端口一起使用。Pynq 中的 DMA 支持 AXI 直接内存访问 IP，允许其从 DRAM 中读取写入数据。

Zynq 支持三种 AXI 总线，分别为：

(4) AXI4：用于高性能内存映射方式的通信需求，允许最大 256 的数据突发传输。

(5) AXI4-Lite：用于简单的低吞吐率的内存映射方式通信需求，是一个轻量级的地址映射单次传输接口，占用很少的逻辑单元。

(6) AXI4-Stream：用于高速流数据传输，允许无限制的数据突发传输模式。

从图 4-30 中可知：

PL 端有 4 个 AXI Master HP 接口、2 个 AXI Master GP 接口、2 个 AXI Slave GP 接口和 1 个 ACP Master 接口。

PS 端有 4 个 AXI Slave HP 接口，2 个 AXI Master GP 接口、2 个 AXI Slave GP 接口和 1 个 ACP Slave 接口。

因此，与监控模块(PS)的连接主要是如何将用户设计的模块的 I/O 与 PS 端的几种接口进行连接的问题。

本书使用 GPIO 来连接 PS 和 PL。Zynq 中有两种实现 GPIO 的方式：一种是使用 PS GPIO(分为 MIO 和 EMIO 两种)，另外一种是在 PL 中加入 AXI_GPIO IP 核(IP 方式)。本书使用的是 AXI_GPIO IP。

- MIO 方式：Zynq 7000 系列芯片有 54 个 MIO，它们分配在 GPIO 的 Bank0 和

Bank1，对 PL 透明。

- EMIO 方式：EMIO 有 64 个引脚可以使用，它们分配在 Bank2 和 Bank3 上，和 PL 相连，如图 4-30 所示。无论 MIO 还是 EMIO 都属于 PS 上的 IO，直接由 PS 操作。
- IP 方式：AXI_GPIO 是通过 AXI 总线挂在 PS 上的 GPIO，PS 端通过 M_AXI_GP 接口来控制 PL 端 AXI_GPIO IP 模块，调用时占用相应 AXI 总线地址空间。

虽然 IP 方式占用 PL 资源，但是其优点是引脚资源丰富和使用灵活方便。因此本书使用 IP 方式实现 GPIO。AXI_GPIO 模块见图 4-31，每个 AXI_GPIO 最多可以有两个通道，每个通道最多有 32 个引脚。AXI_GPIO 左端与 PS 通过 AXI 总线相连，其右端与用户模块相连。

第 5 章"基于 Python 的 I/O 交互"将会进一步介绍如何采用 Python 编码通过对 PS 端接口的控制完成对 PL 端用户模块的输入和输出。



图 4-31 AXI_GPIO 模块

2. 与 PS 的连接

在基于 PYNQ 的计算机组成原理实验中，最直接的方式是将用户模块通过 GPIO 与 PS 端连接，并通过在 PS 端运行 Python 对 GPIO 进行操作完成对 PL 中用户模块的输入/输出。具体介绍如下：

在完成用户模块设计，需要将其与 PS 进行连接时，单击左侧 Create Block Design 打开新建原理图对话框，如图 4-32 所示。

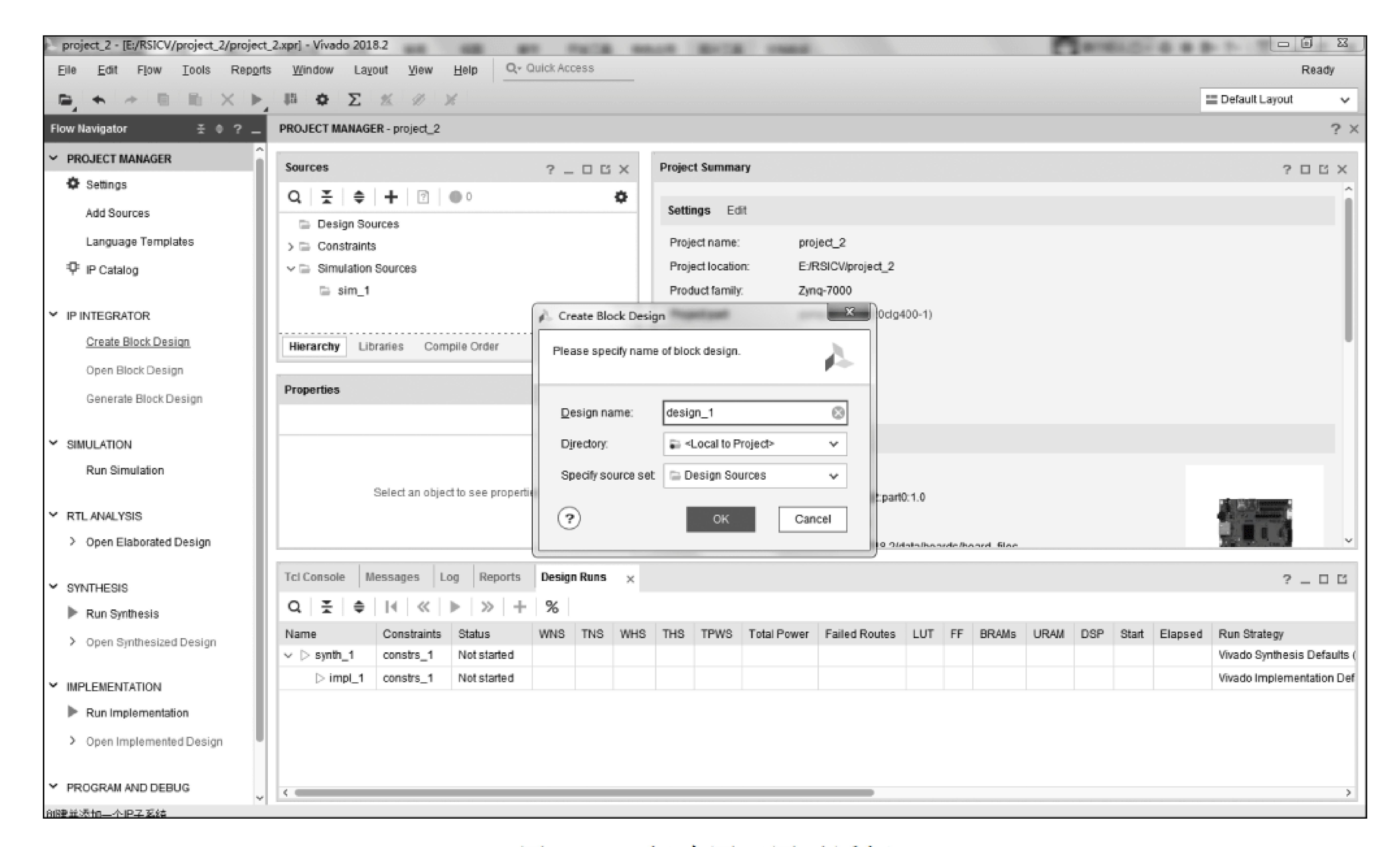


图 4-32 新建原理图对话框

如图 4-33 所示，右击，选择 Add IP 输入 ZYNQ7 Processing System，添加 ZYNQ7 Processing System IP 核。它代表的是 Zynq7000 SoC 芯片中的 PS 端。如图 4-34 所示，在此调出后可以以原理图的方式操纵它与用户模块的连接，当然还需要添加其他中间模块。

单击 Run Block Automation，自动将 DDR 与 FIXED_IO 进行配置，并将 PYNQ 的

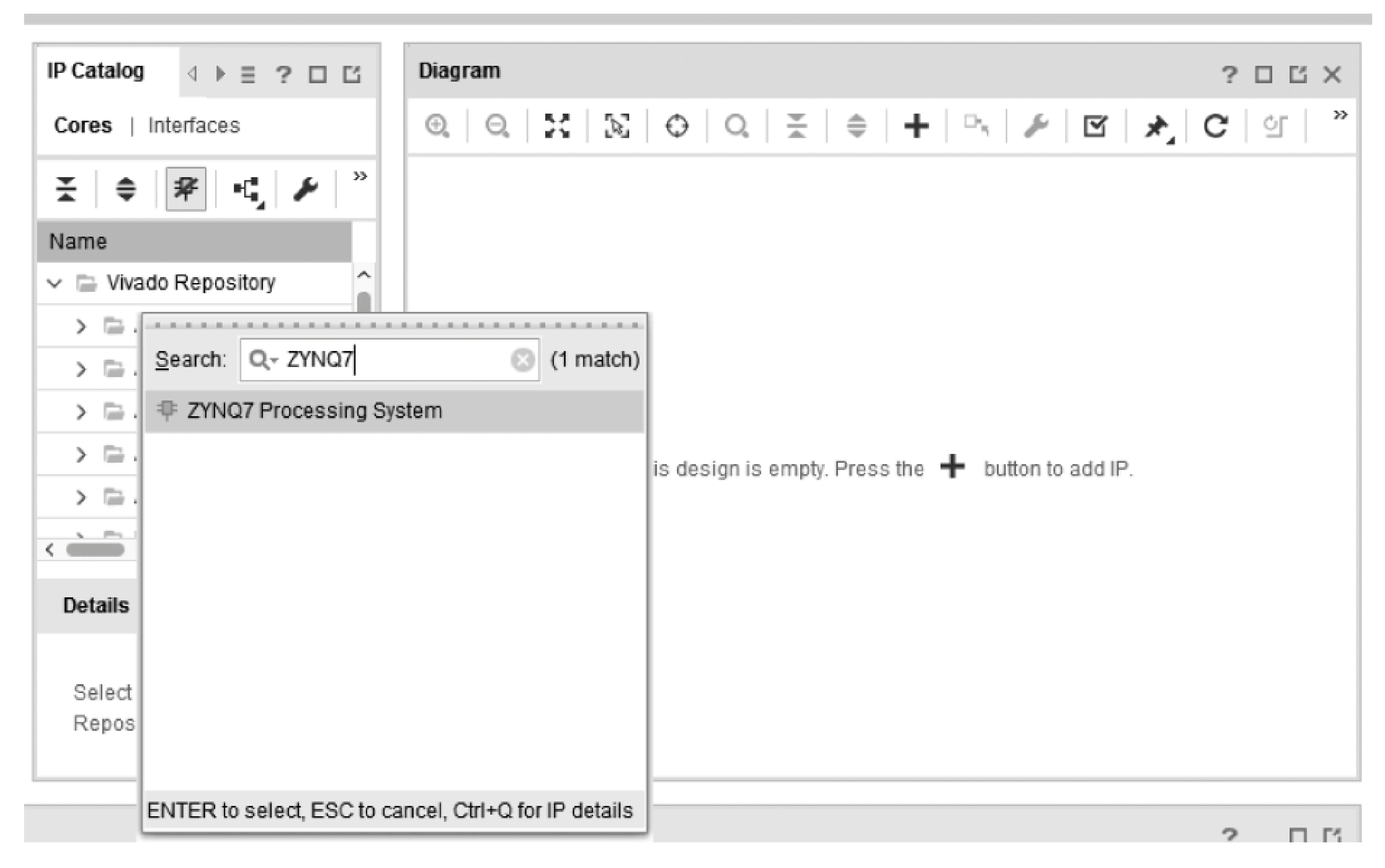


图 4-33　添加 ZYNQ7 Processing System IP

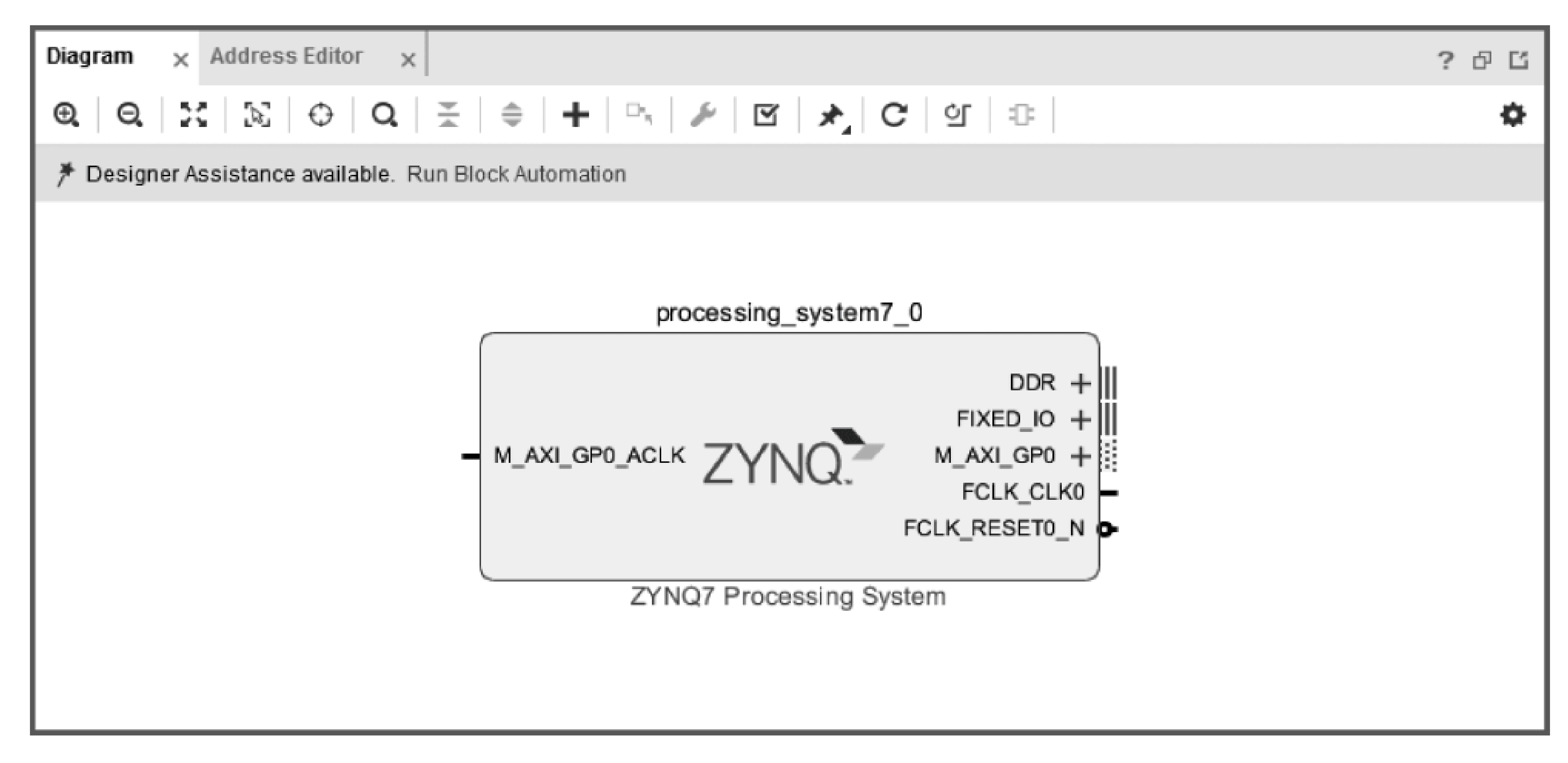


图 4-34　ZYNQ7 Processing System 的原理图方式

ZYNQ Block Design 进行默认配置。

右击，选择 Add IP 添加 AXI GPIO 模块。如图 4-35 所示，用来提供与 PL 上的用户硬件模块连接的 GPIO 接口。PS 的 M_AXI_GP0 与 AXI GPIO 模块的 S_AXI 相连，也就是说 AXI GPIO 是通过 PG 通道来连接的。AXI GPIO 模块的 GPIO 引脚与 PL 中的用户模块相连。

Run Block Automation 是 Vivado 根据接口进行的自动连接。自动连接时可以选择相应的接口，这适用于简单的连接。当遇到复杂的设计时，如多个时钟输入，为了确保设计的正确性，可能需要进行手动连线并检查连接的正确性。

在图 4-35 中，GPIO 模块上有一个 AXI Slave 接口。如果要将 PS 与 AXI GPIO 模块相连接，需要在 PS 端的 GP 口配置一个 AXI Master 接口。在图 4-34 中，ZYNQ7 Processing

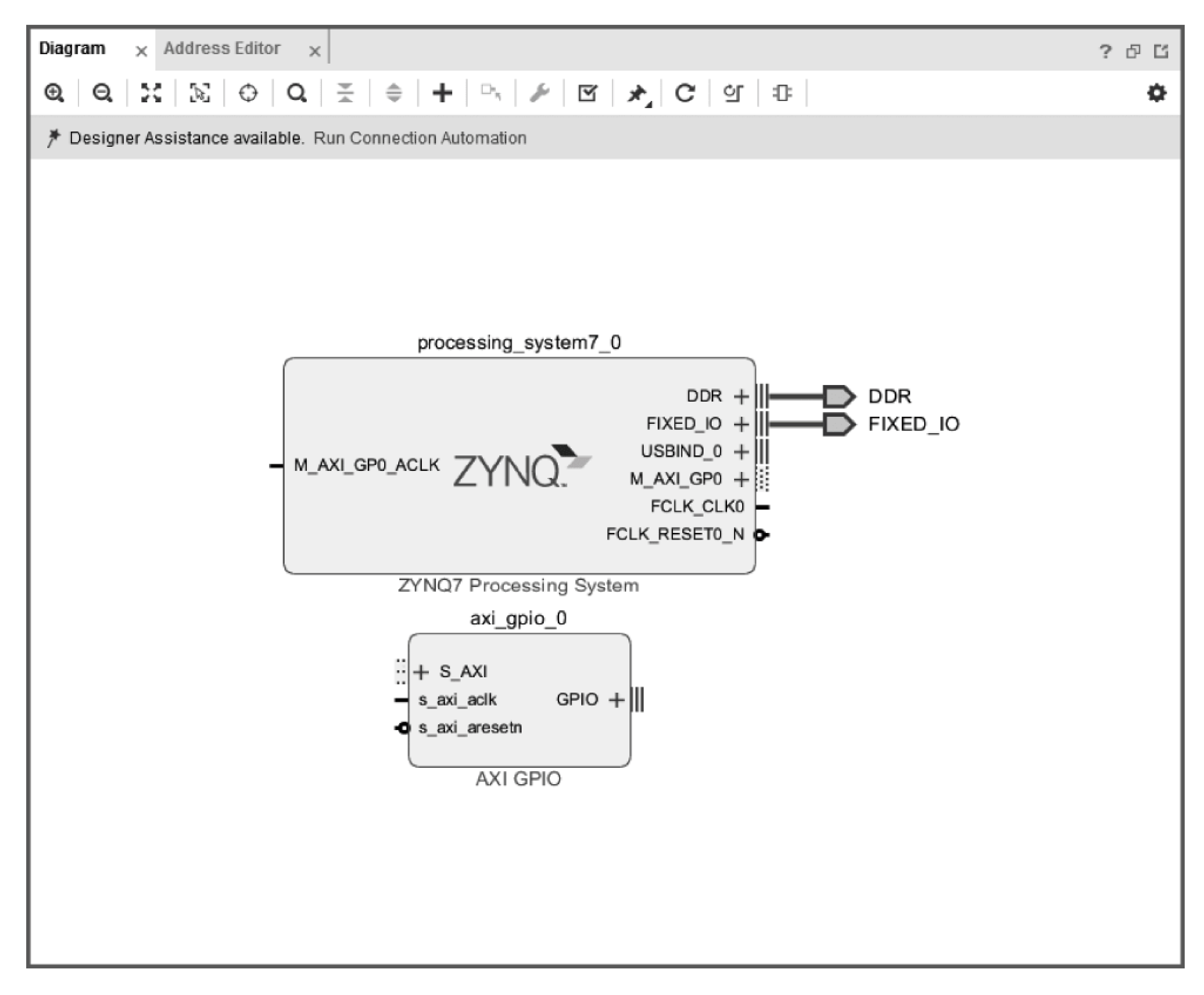


图 4-35 增加 GPIO 模块

System IP 核已经默认配置了一个 AXI Master 接口(M_AXI_GP0),因此这里无须重复配置。

如需配置其他参数,可双击图 3-34 中的 ZYNQ7 Processing System IP 核,即打开图 4-36 所示界面。在此可以进行 PS-PL 的连接方式配置(其中包含常用的 GP、HP 口等)、物理 I/O 引脚配置、MIO 配置、时钟配置、SMC 时钟计算、中断配置等。

选择图 4-36 左侧的 PS-PL Configuration 可进行 PS-PL 连接方式的配置(见图 4-37),M_AXI_GP0 interface 被勾选,说明当前 PS 和 PL 之间可通过 M_AXI_GP0 进行连接。图 4-35 中也对应显示了 ZYNQ7 Processing System 模块的 M_AXI_GP0 通道是可用的。未被勾选的通道在图 4-35 中就没有对应显示。

选择图 4-36 左侧的 Clock Configuration 可进行时钟设置。如图 4-38 中,通过勾选设置 FCLK_CLK0 为 PL 中的功能模块所使用的时钟,相应地在图 4-35 中就会显示 ZYNQ7 Processing System 模块有 FCLK_CLK0 的时钟输出。此处可以设置时钟源及频率,如 PLL/100MHz。

选择 Run Connection Automation,右击,弹出如图 4-39 所示对话框。将 GPIO 模块中 S_AXI 的三个时钟接口都连接到 FCLK_CLK0 上。由于之前配置了 Pynq-Z2 的 Boardfiles,AXI GPIO 模块的 GPIO 引脚除了可连接用户模块,也可连接 PYNQ 板卡的任何 GPIO 资源。例如,这里选择 Leds 4bits,GPIO 会自动配置为 4 位的输出。然后单击 OK 按钮,进行自动连接。

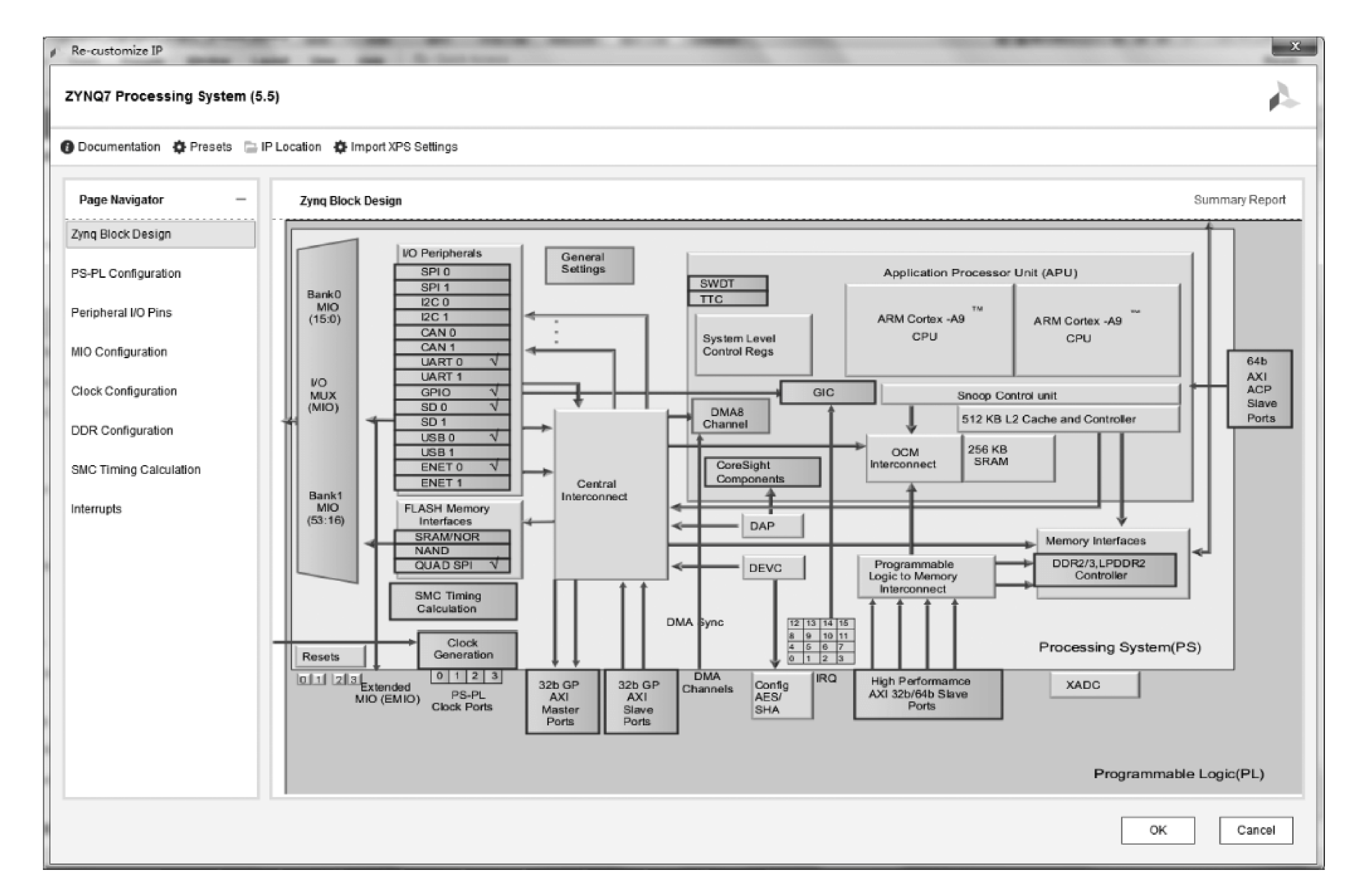


图 4-36　ZYNQ7 Processing System 配置界面

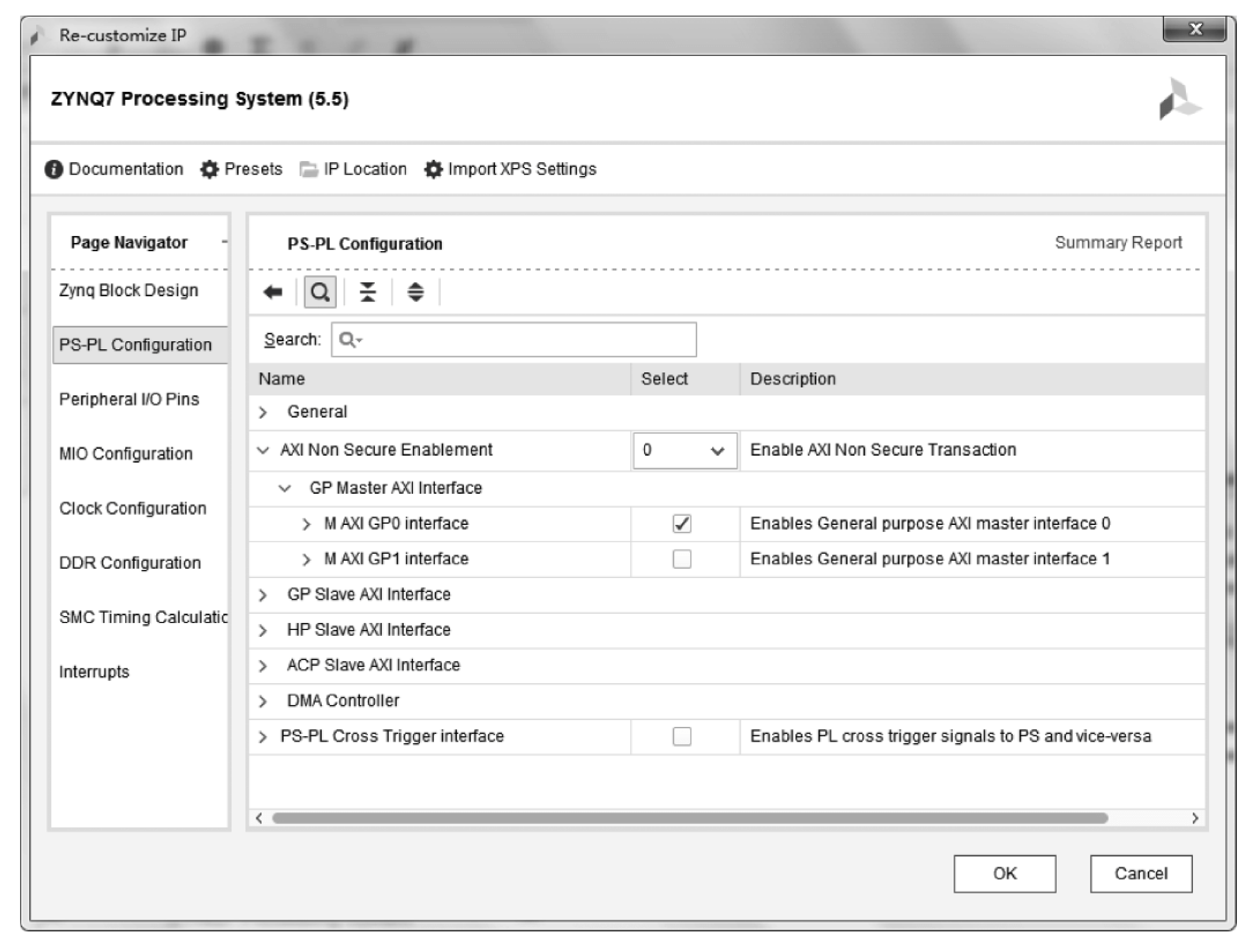


图 4-37　PS-PL 连接方式配置

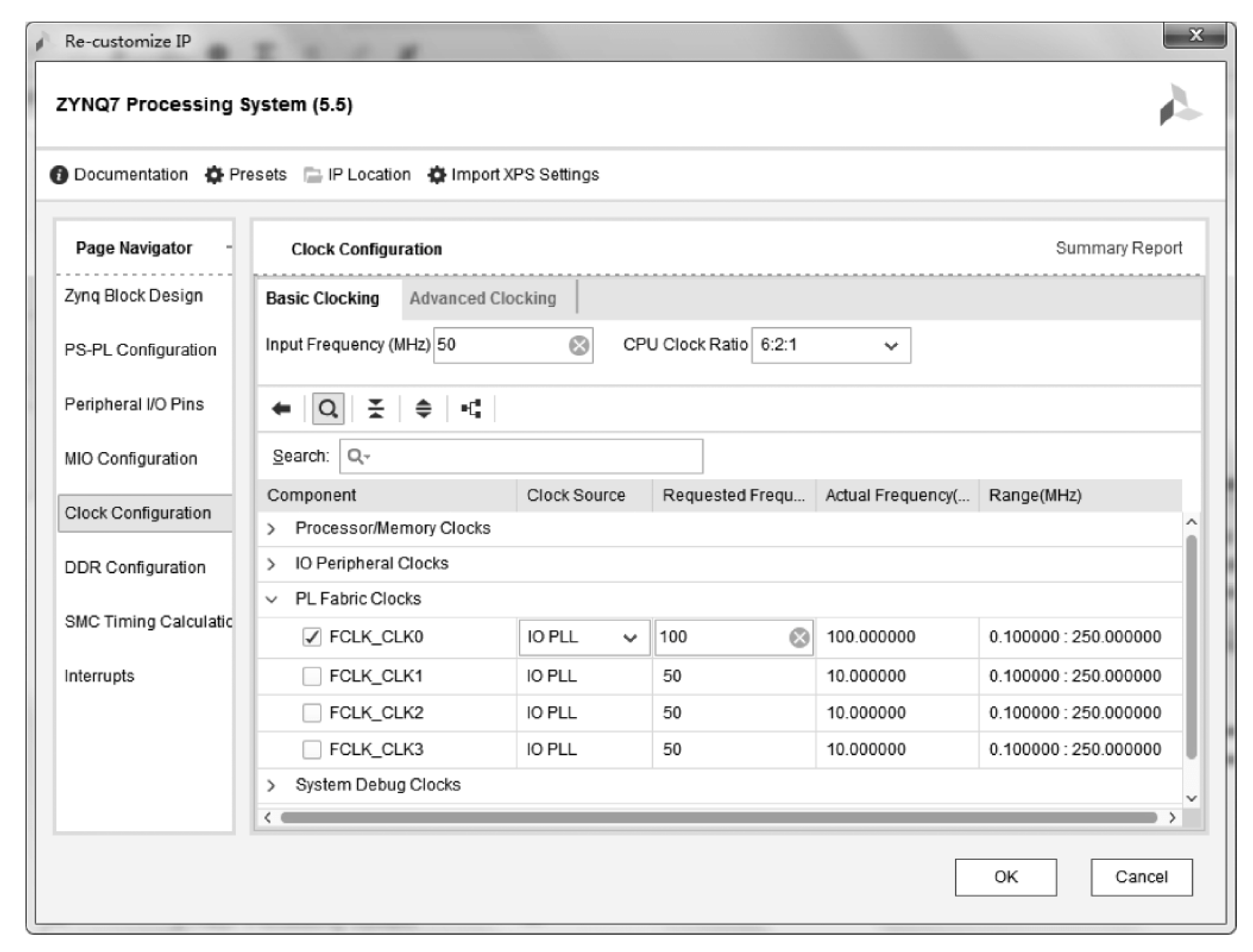


图 4-38　时钟设置

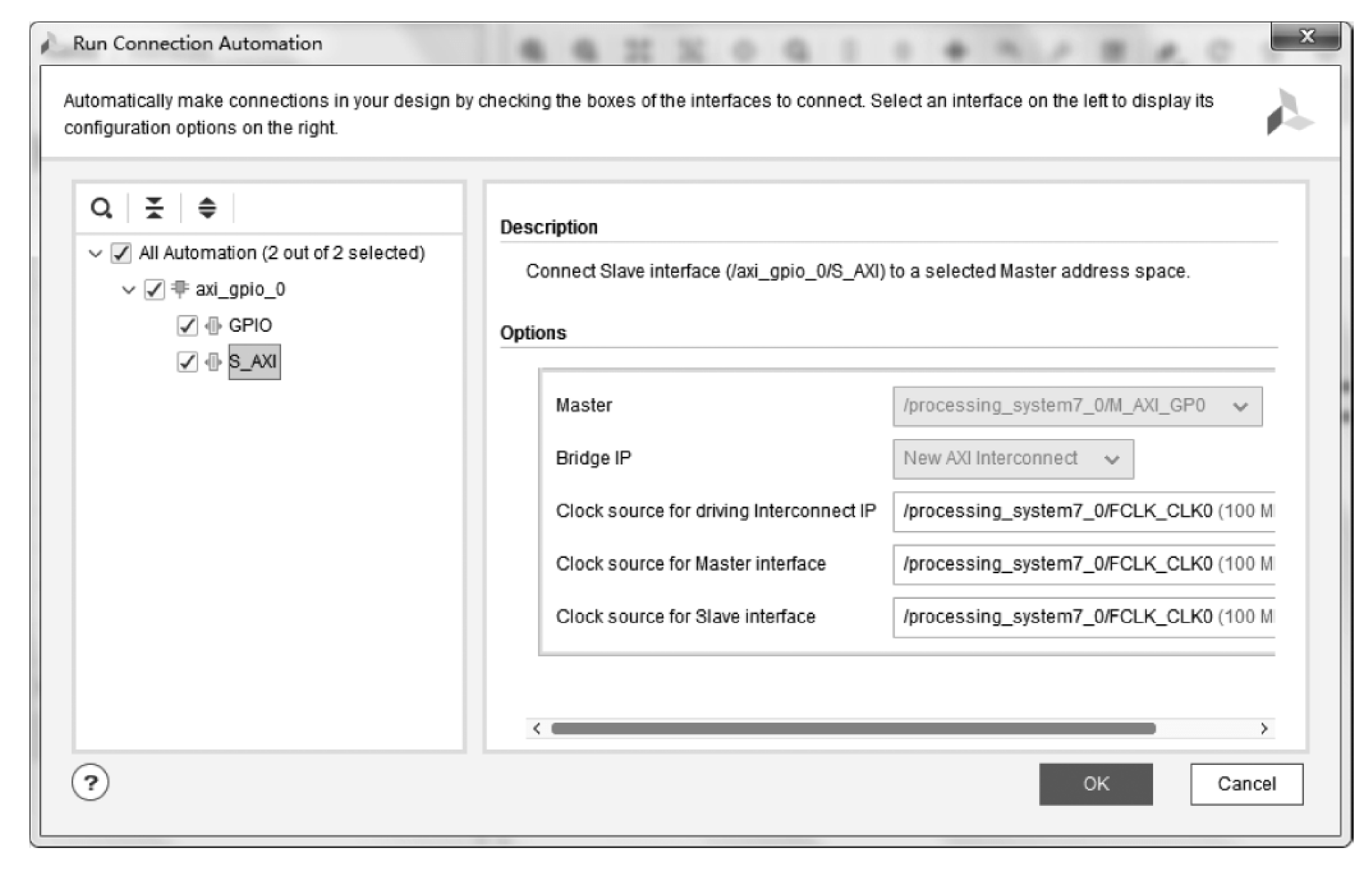


图 4-39　Run Connection Automation 配置

自动连接后可右击，选择 Regenerate Layout 对 Block Design 进行自动布局，布局完成后显示如图 4-40 所示的连接关系。

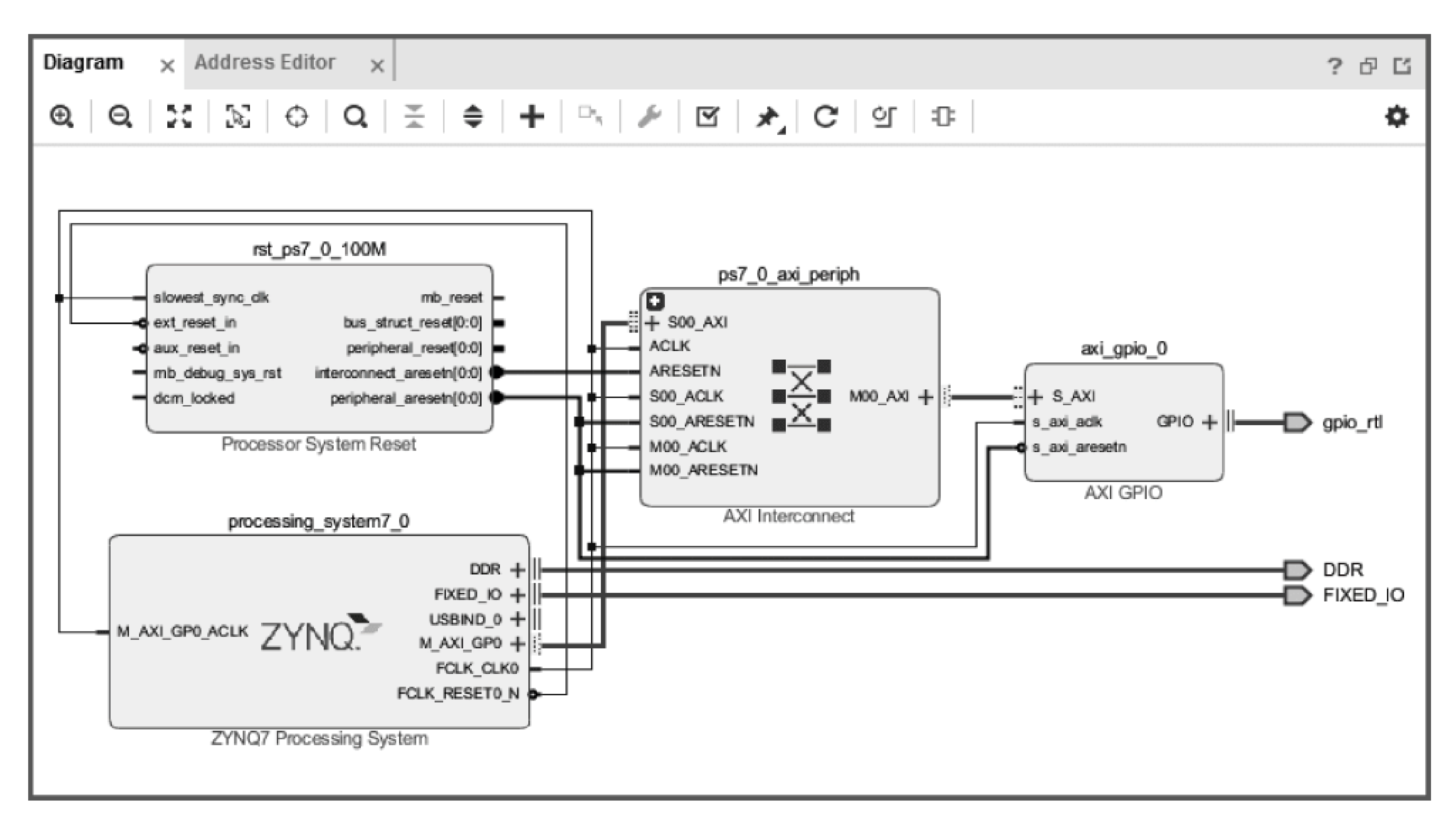


图 4-40 PS 与 PL 的连接关系

从图 4-40 可知，Run Connection Automation 还会自动加载连接到所需的其他必要的模块，如 Processor System Reset、AXI Interconnect 模块。

AXI GPIO 模块的 S_AXI 端口与 PS 连接，而其 GPIO 引脚可以与 PL 端的用户模块连接。运行在 PS 上的 Python 代码就可以通过控制 GPIO 引脚完成与用户硬件模块的交互。另外，GPIO 引脚也可以直接连接到板卡上的外设，运行在 PS 上的 Python 代码就可以通过控制 GPIO 引脚完成板卡上外设的控制和交互。

4.6 实现

实现主要是完成转换、映射、布局布线和最终比特流的生成。意思是将综合生成的网表一步步地与特定的 FPGA 芯片关联起来，转换成该 FPGA 芯片上特有的资源并进行映射。实现后的时钟才是与具体期间匹配之后得到的更准确的时钟。

选择 Flow Navigator 中的 IMPLEMENTATION→Run Implementation 可进行布局布线等实现过程。可从 Reports 中查看 Opt Design、Place Design、Route Design 等报告。当然对于计算机组成原理实验来说，实现这一步骤通常是由 Vivado 自动完成，许多内部的细节并不需要深究，除非在此过程中发生错误时需要根据提示进行相应处理。

Flow Navigator 中的 IMPLEMENTATION→Open Implementation Design 可查看实现之后的报告。

想要将设计加载到 FPGA 片上，需要生成比特流文件。生成比特流文件很简单，只需要选择 Flow Navigator 中的 PROGRAM AND DEBUG→Generate Bitstream 便可生成。

在基于 PYNQ 的组成原理实验中，通常采用 Python 代码的方式加载并运行比特流文件，因此，这里就不再介绍比特流文件的下载过程。在后续章节的 Python 代码示例中会介

绍如何加载生成的比特流并运行。

4.7　TCL 使用介绍

TCL(Tool Command Language)是一种由 John Ousterhout 创建的脚本语言，最早称为“工具命令语言”。TCL 易学易用且功能强大，经常被用于快速原型开发、脚本编程、GUI 和测试等方面。

在 Vivado 中也提供了 TCL 命令。利用 TCL 提供的标准语法，用户可直接发送指令到 Vivado 中。可利用 TCL 创建、打开、保存工程，找某个时钟的 BUFG 或者综合后网表中的某一个信号，自动触发 ILA 抓取信号等。

为了使用 PYNQ 中 Overlay 的方式与 PL 进行交互，需要了解更多 Block Design 的信息，所以需要生成有关 Block Design 信息的 TCL 文件。

如图 4-41 所示，打开 Block Design 页面，选择 File→Export→Export Block Design 生成. tcl 文件。

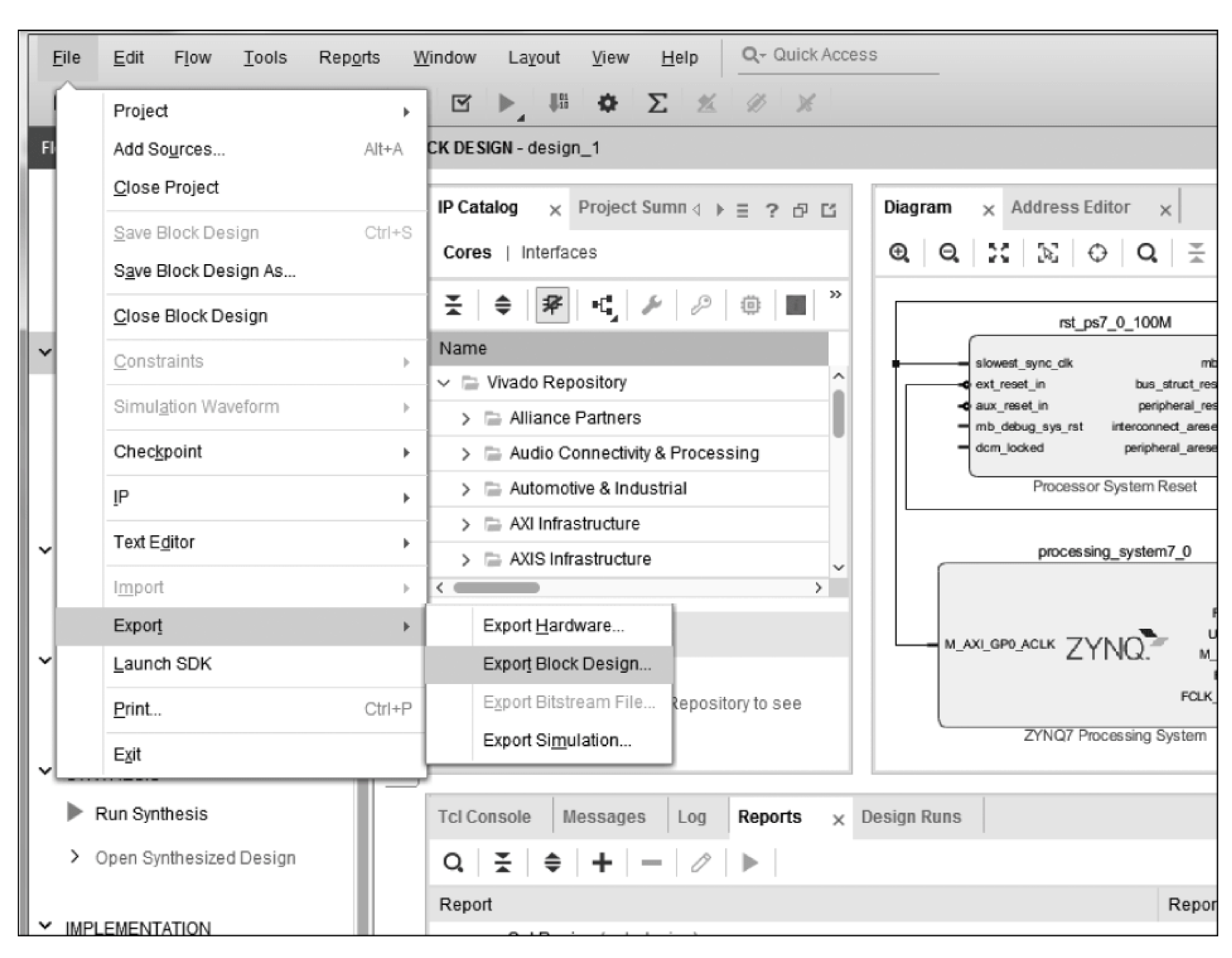


图 4-41　生成 tcl 文件

. tcl 文件还可用于还原整个工程，避免重复工作。可选择 Tool→Run Tcl Script，如图 4-42 所示。

由于在后续章节的实验中经常会用到部分相同的原理图设计，如果每次实验时都要添加各个模块并进行模块间连接是没有必要的。为了避免大量重复的工作，可以将这部分原理图设计连接完成并将其生成一个 tcl 文件。当下一次使用时可以直接通过 Vivado 软件的 tcl

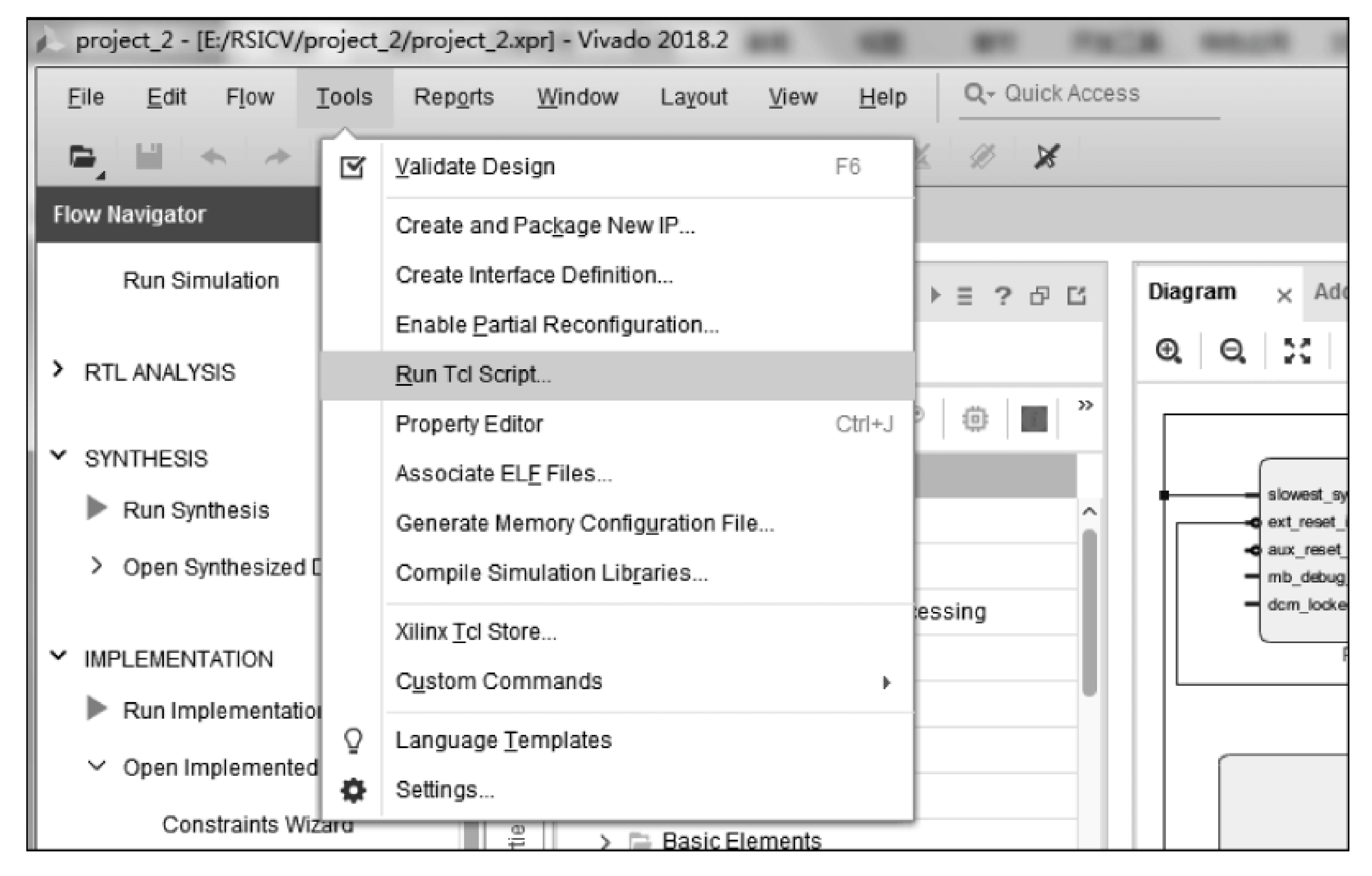


图 4-42　运行 tcl 文件

console 面板打开相关的 tcl 文件后将之前连接好的部分原理图设计调用出来，避免了重复操作。tcl console 界面如图 4-43 所示，通过在方框处输入打开 tcl 的语句，来实现原理图的调用。

```
Tcl Console  Messages  Log  Reports  Design Runs
# }
# create_root_design ""
create_bd_cell: Time (s): cpu = 00:00:04 ; elapsed = 00:00:05 . Memory (MB): peak = 890.992 ; gain = 54.543
Wrote  : <D:/FPGA/book/project_1/project_1.srcs/sources_1/bd/design_1/design_1.bd>
Wrote  : <D:/FPGA/book/project_1/project_1.srcs/sources_1/bd/design_1/ui/bd_1f5defd0.ui>
update_compile_order -fileset sources_1
Type a Tcl command here
```

图 4-43　tcl console 界面

通过 source 命令打开 tcl 文件，输入 tcl 文件路径调用原理图，如图 4-44 所示。

```
#
# } else {
#    # USE CASES:
#    #    8) No opened design, design_name not in project.
#    #    9) Current opened design, has components, but diff names, design_name not in project.
#
source d:/e203/e203/decode/寄存器/调试/interface.tcl
```

图 4-44　原理图调用语句

4.8　实例演示

接下来再通过一个简单的例子($S=\overline{A}B$)来展示上述的完整 Vivado 开发流程，上传比特流到 PYNQ 上并进行 Python 交互的部分将会在第 5 章介绍。

4.8.1 原理图方式

原理图方式包括创建工程、设计输入、仿真、与 PS 的连接、综合、实现、生成比特流等。

1. 创建工程

参照 4.1 节在 boards 列表中选择 PYNQ_Z2，完成工程(project_1)的创建。

2. 设计输入

单击 project_1 界面左侧的 IP INTEGRATOR 下的 Creat Block Design，进入原理图设计界面，如图 4-45 所示。

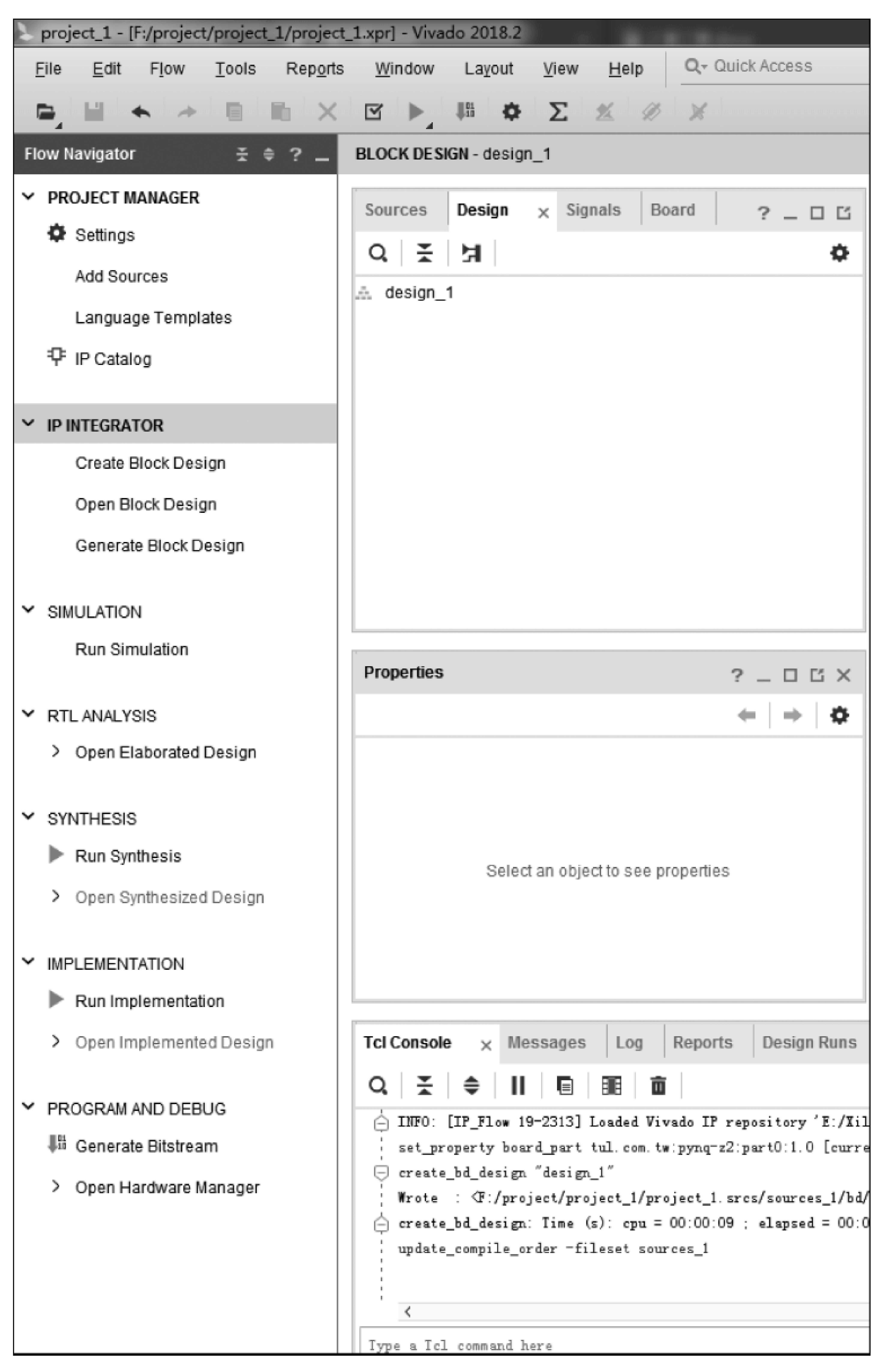


图 4-45 原理图设计

可以添加自带的 IP 或用户手动生成的 IP 来完成设计，用户可手动生成两个简单的 IP，AND 和 NOT。首先再重新创建一个工程(project 2)，右击，选择 Design Source 选择 Add Sources，再选择 Create File，分别创建名为 AND. v 和 NOT. v 的 Verilog HDL 源文件。

AND. v 代码如下：

```
module AND(
    input A,
    input B,
    output S );
assign S = A & B;
endmodule
```

NOT. v 代码如下：

```
module NOT(
    input A,
    output S
          );
assign S = ~A;
endmodule
```

在 project_2\project_2. srcs\sources_1\new 下创建两个文件夹 IP_AND 和 IP_NOT(见图 4-46)，再将 AND. v 和 NOT. v 分别复制到 IP_AND 和 IP_NOT 中去。

名称	修改日期	类型	大小
IP_AND	2019/4/9 19:25	文件夹	
IP_NOT	2019/4/9 19:25	文件夹	
AND.v	2019/4/9 19:16	V 文件	1 KB
NOT.v	2019/4/9 19:15	V 文件	1 KB

图 4-46　文件夹创建

在工程主界面将 AND. v 设置为顶层文件，单击 Tools，选择 Create and Package New IP，在 Packaging Options 中选择 Package a specified directory，出现如图 4-47 所示文件夹。

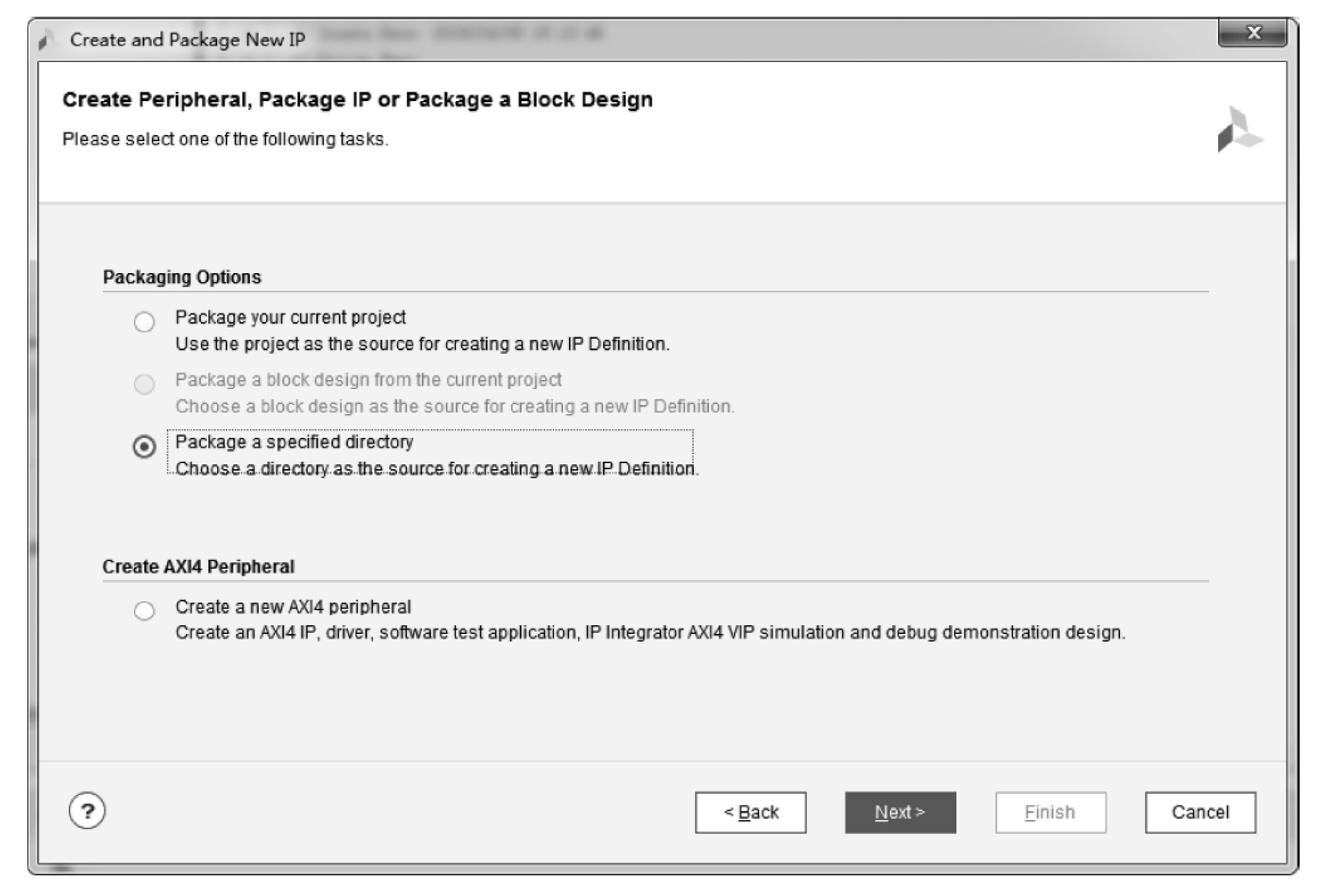


图 4-47　打包一个文件夹

在 IP 生成地址中选择刚才创建的 IP_AND 文件夹(见图 4-48),打包成 IP 核。

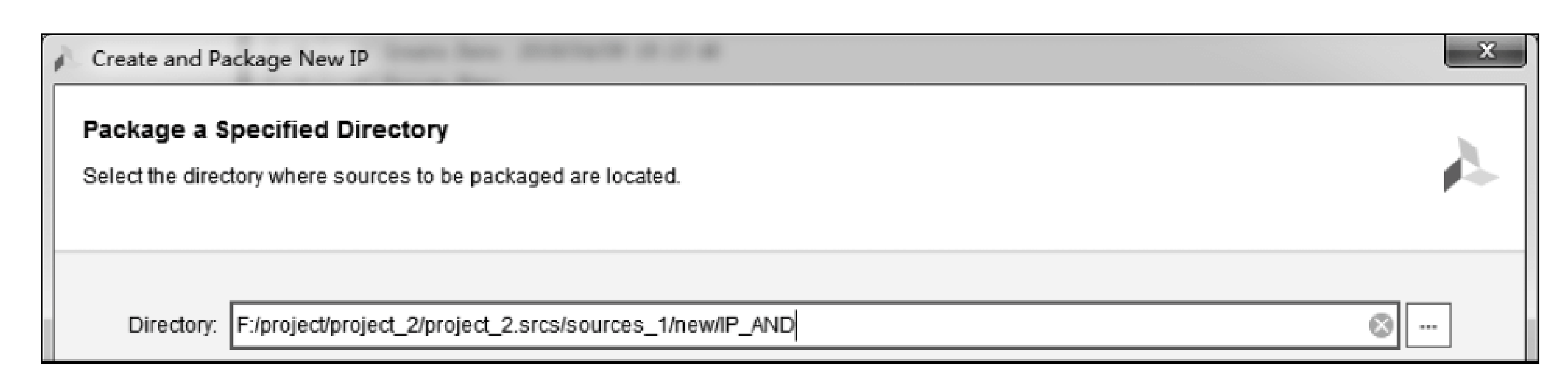


图 4-48　打包成 IP

同理,将 NOT. v 设置为顶层文件后,在 IP_NOT 文件夹中生成 NOT. v 的 IP。

在 project_1 的 Project Settings 中单击 IP Repository 添加 IP 的存放路径(见图 4-49),单击 Apply 按钮,将 IP 包添加到工程中。

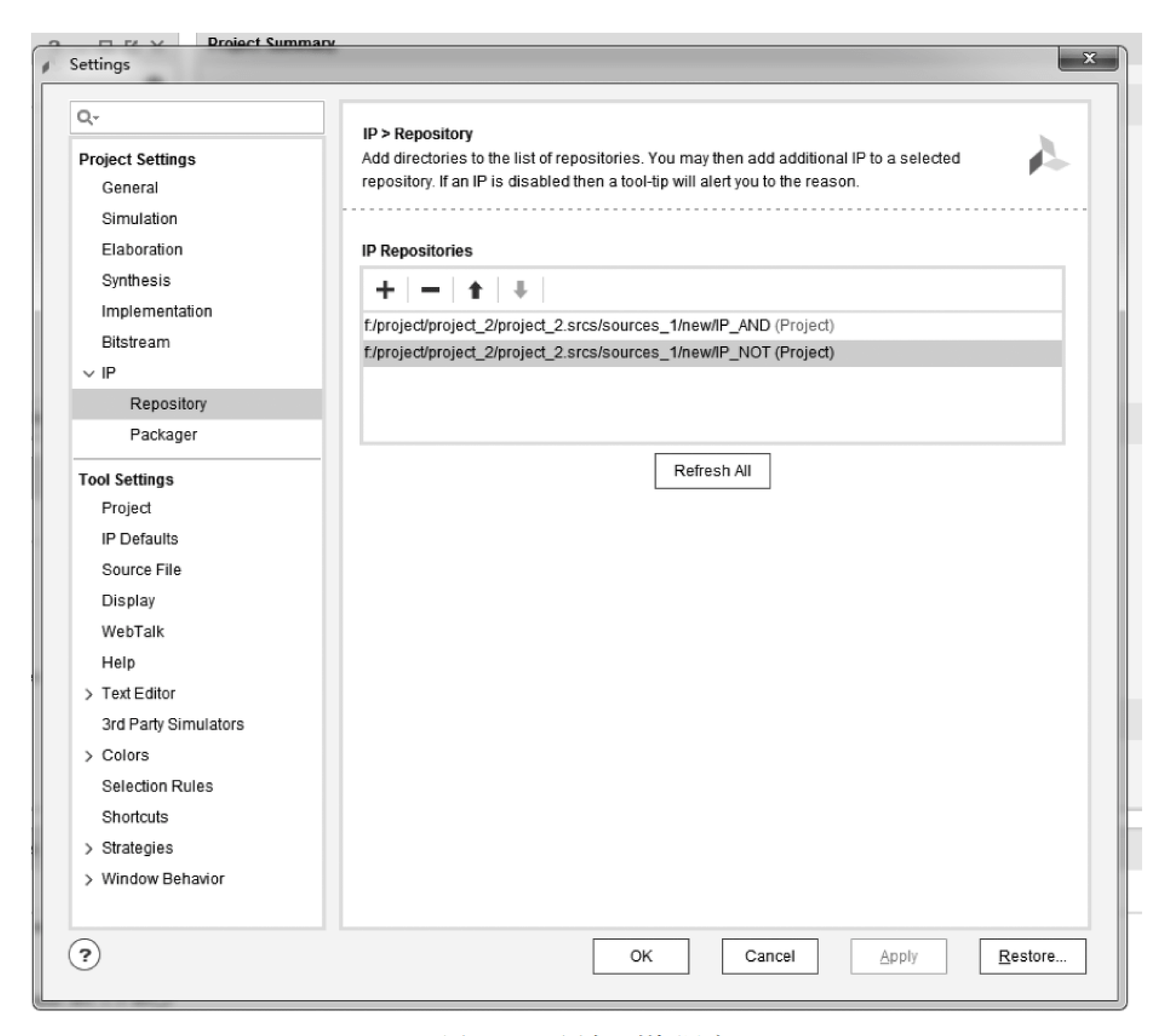


图 4-49　添加到资源库

在 project_1 的 Block Design 中添加这两个 IP。参照 3. 2. 1 节完成引脚引出和 HDL 封装,如图 4-50 所示。

3. 仿真

对于 HDL 封装形成的 HDL wrapper 可以进行仿真。在 project_1 主界面右击,选择 Simulation Source,选择 Add Source(见图 4-51),然后选择 Add or create simulation sources,再选择 Create File,最后给 Testbench 命名,如图 4-52 所示。

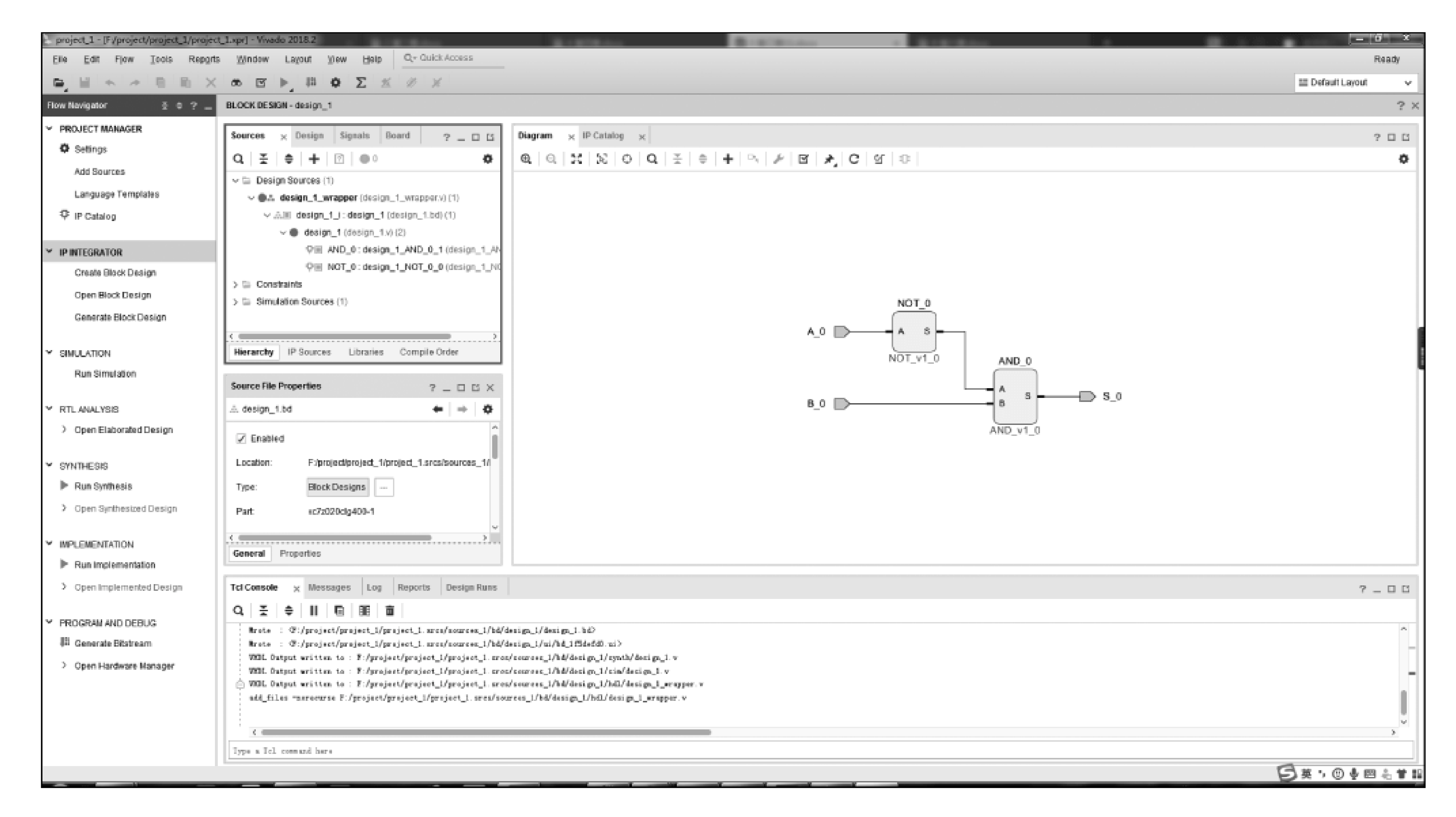


图 4-50　原理图界面

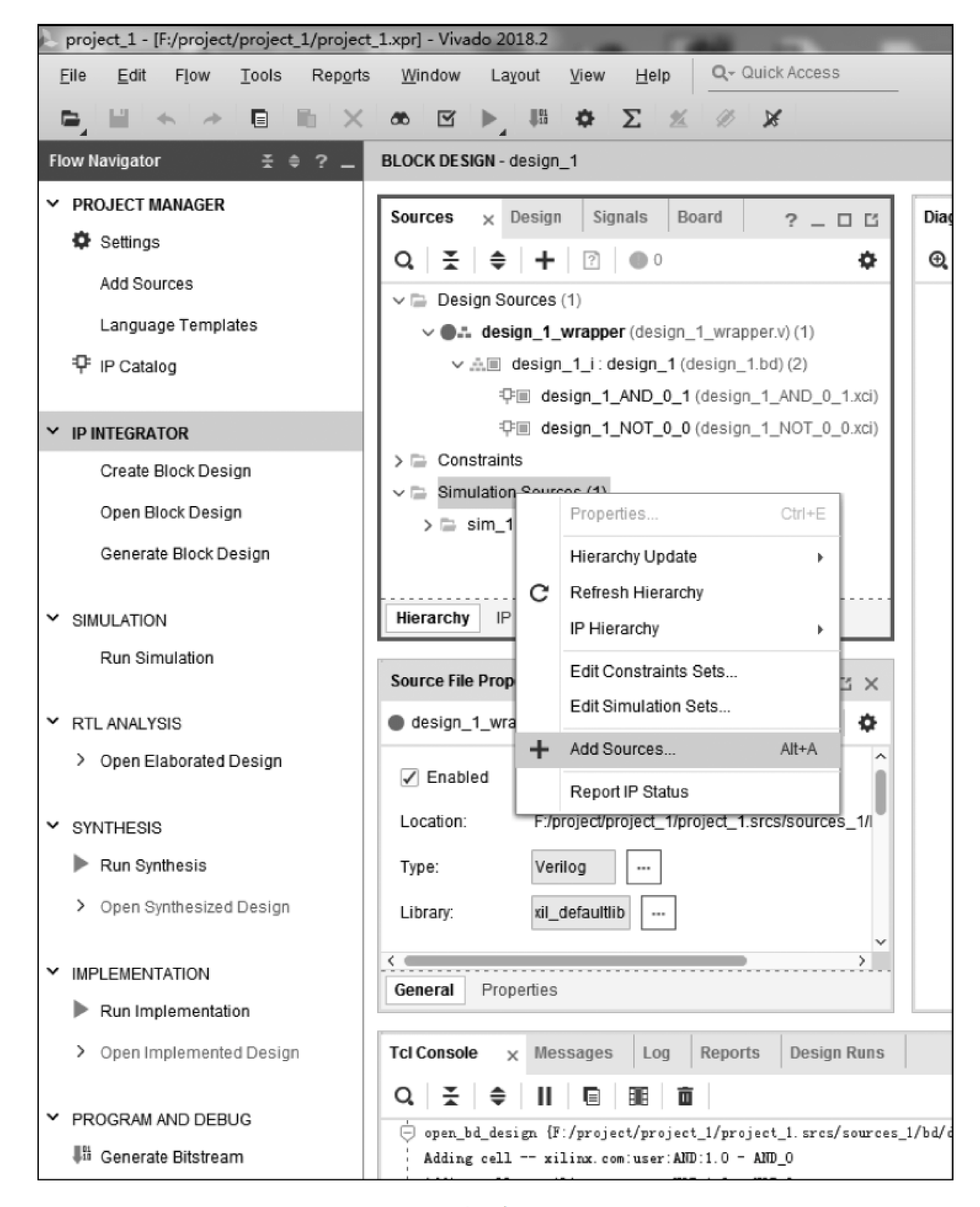


图 4-51　创建 Testbench

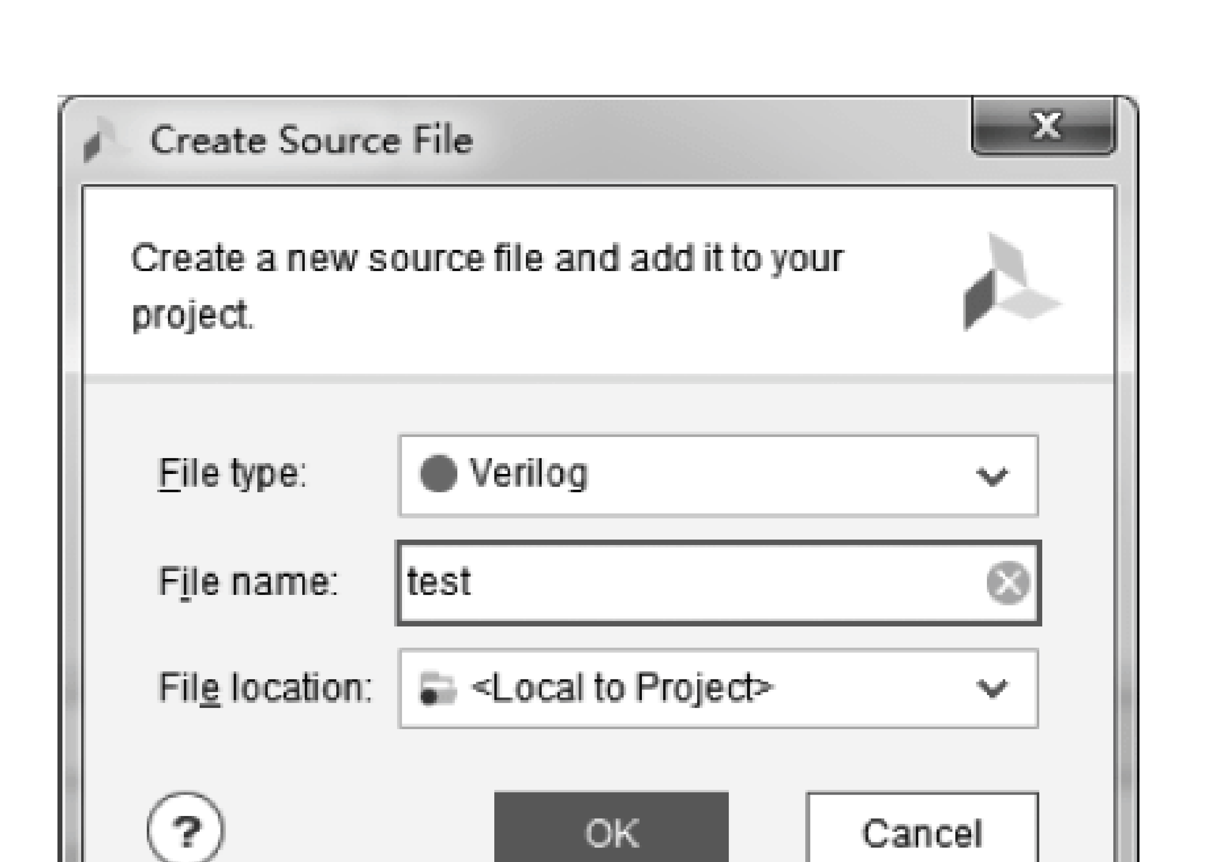


图 4-52　Testbench 命名

Testbench 示例代码如下：

```
module test( );                 //Testbench 模块名,为顶层模块,且没有输入/输出
    reg A;
    reg B;
    wire S;
    design_1_wrapper u1(        //例化 design_1_wrapper
        .A_0(A),
        .B_0(B),
        .S_0(S)
        );
    initial
    begin
        #20                     //延时 20 个时间单位
        A = 1'b1;
        B = 1'b0;
        #20                     //延时 20 个时间单位
        A = 1'b0;
        B = 1'b0;
        #20                     //延时 20 个时间单位
        A = 1'b0;
        B = 1'b1;
        #20                     //延时 20 个时间单位
        A = 1'b1;
        B = 1'b1;
    end
endmodule
```

Testbench 编写完成后，在左侧工程管理栏中选择 Simulation→Run Simulation→run behavioral simulation 可进行行为仿真验证，结果如图 4-53 所示。

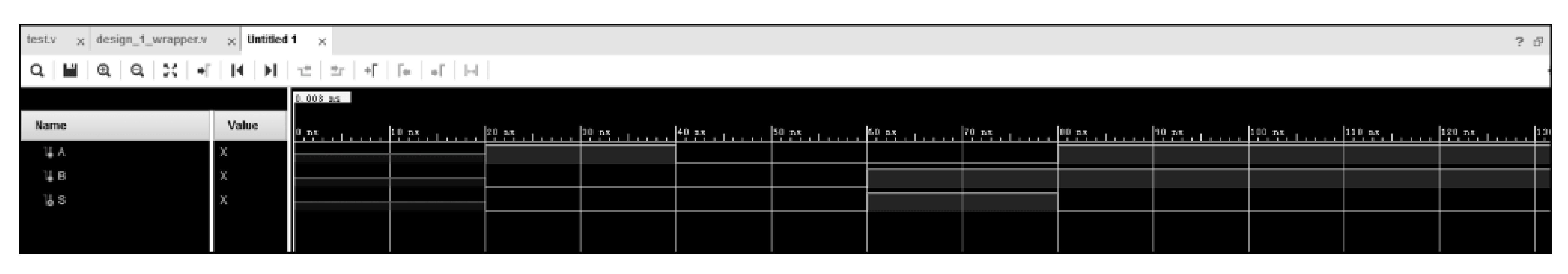


图 4-53　仿真波形

4. 与 PS 的连接

单击 project_2 界面左侧的 IP Integrator 下的 Creat Block Design，创建一个新的 Block Design。参照 4.5.2 节完成对 PS 端的配置，不同的是这里需要添加两个 AXI_GPIO 模块，如图 4-54 所示。删除两个 GPIO 的 arduino_a0_a5 接口。

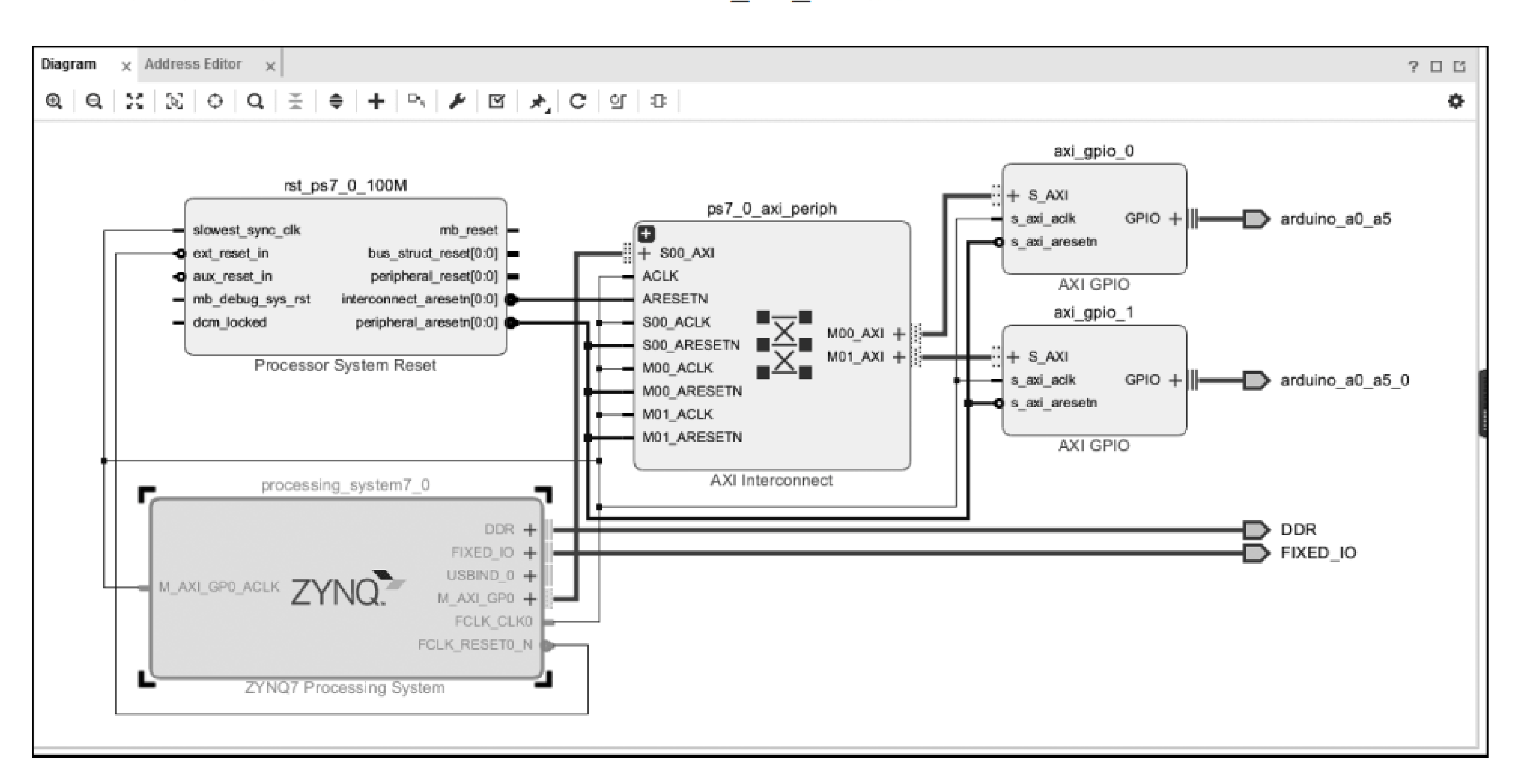


图 4-54 原理图连接

双击 axi_gpio_0，在 Board 中将 Board Interface 设置为 Custom，如图 4-55 所示。在 IP Configuration 中按图 4-56 进行设置，为 GPIO 设置两个 1 位输出端口。

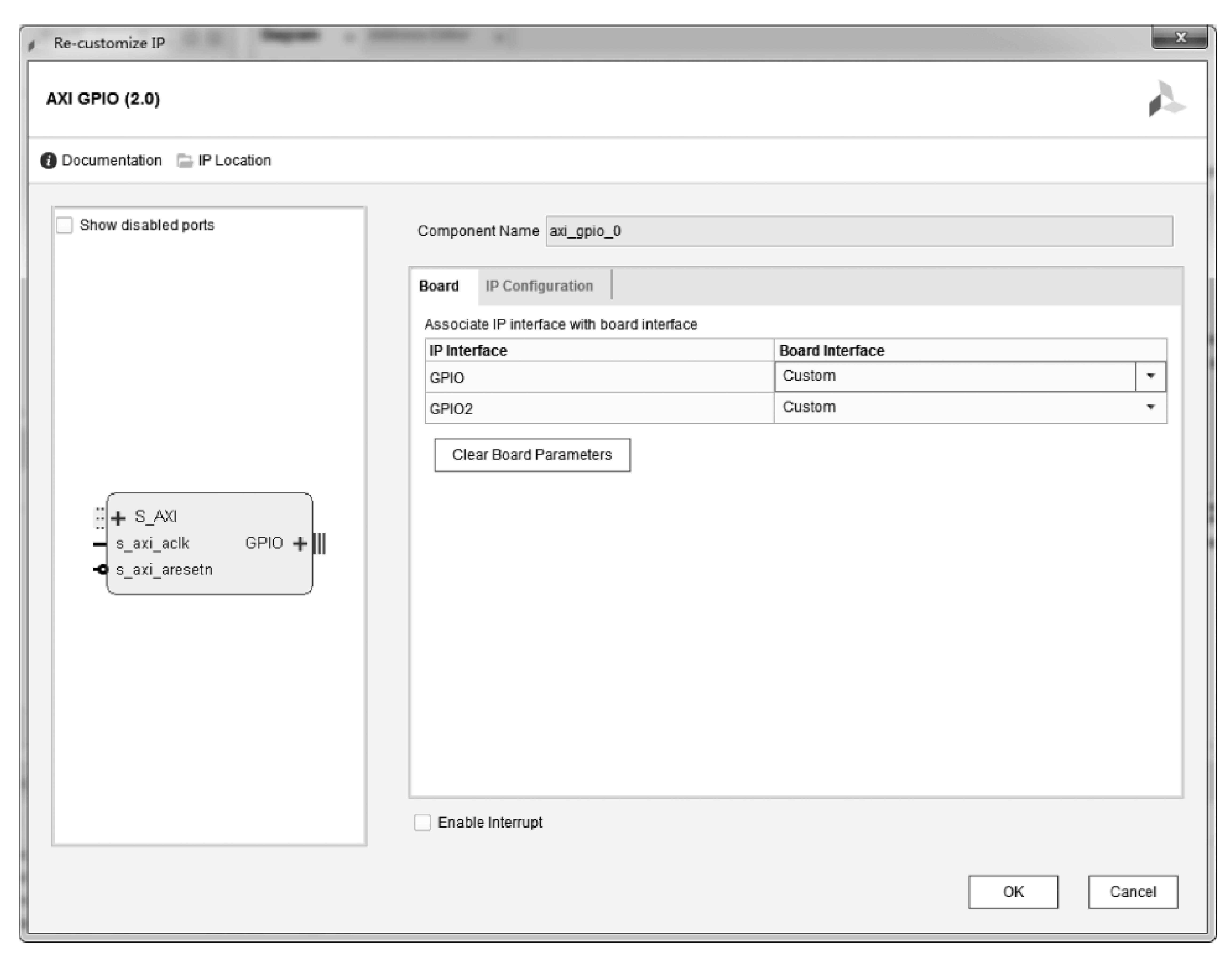


图 4-55 配置 IP 核 1

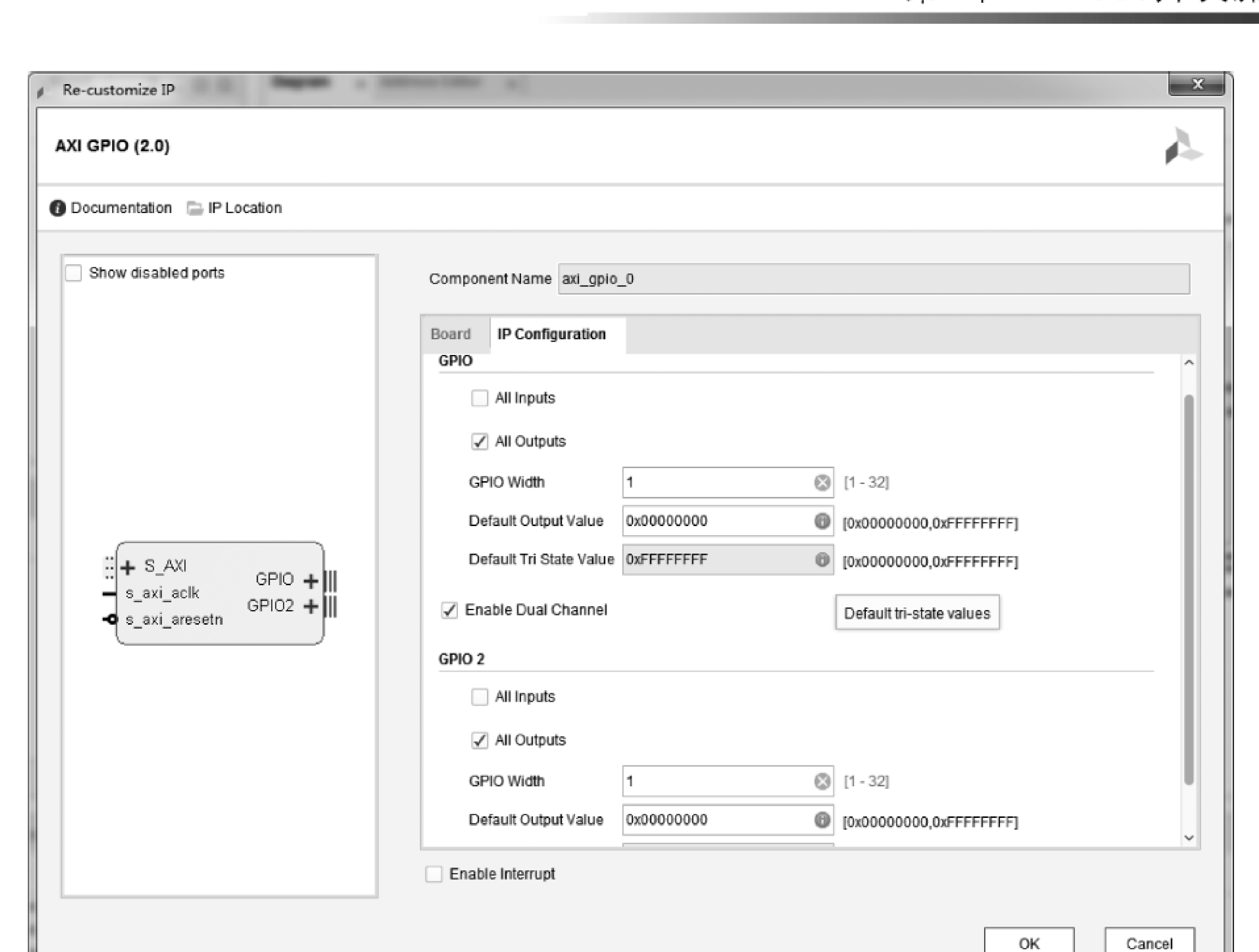


图 4-56 配置 IP 核 2

双击 axi_gpio_1，在 Board 中将 Board Interface 设置为 Custom。在 IP Configuration 中按图 4-57 进行设置，为 GPIO 设置一个 1 位输入端口。

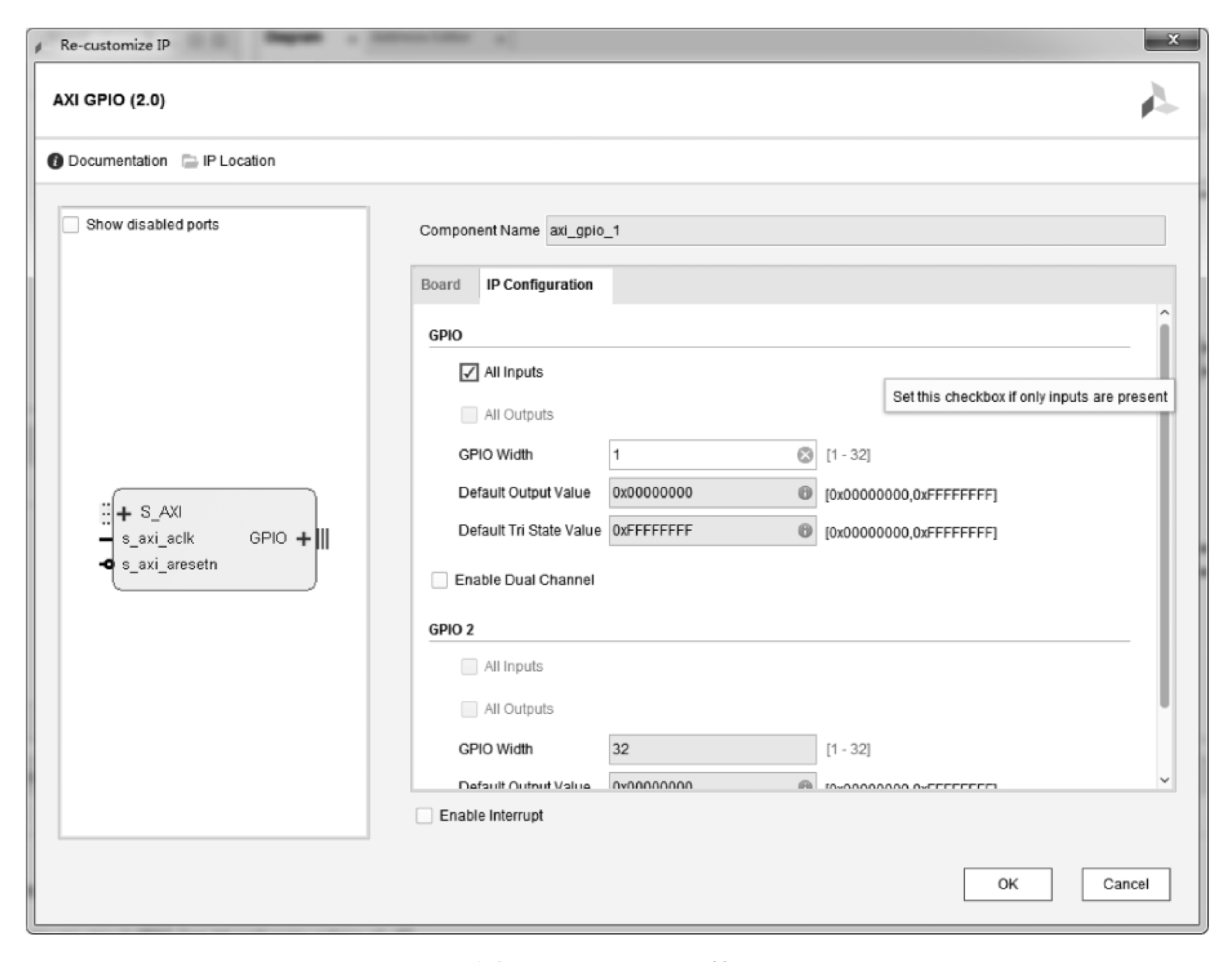


图 4-57 配置 IP 核 3

此时 Block Design 如图 4-58 所示。

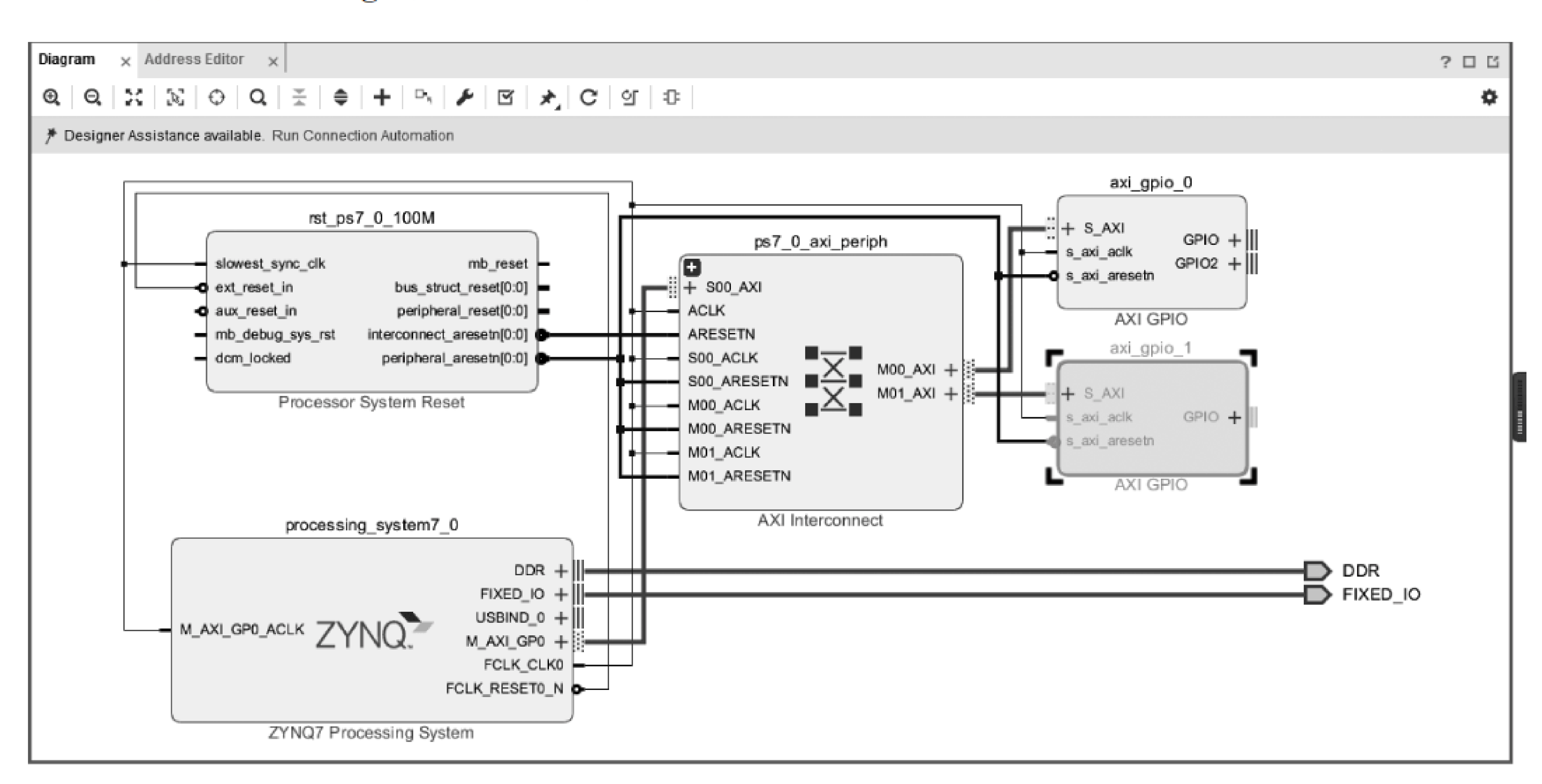


图 4-58　加入了 GPIO 后的最终连接图

添加之前生成好的 AND_v1_0 和 NOT_v1_0 两个 IP，并与两个 GPIO 按图 4-59 进行连接，完成 PS 和 PL 的连接。

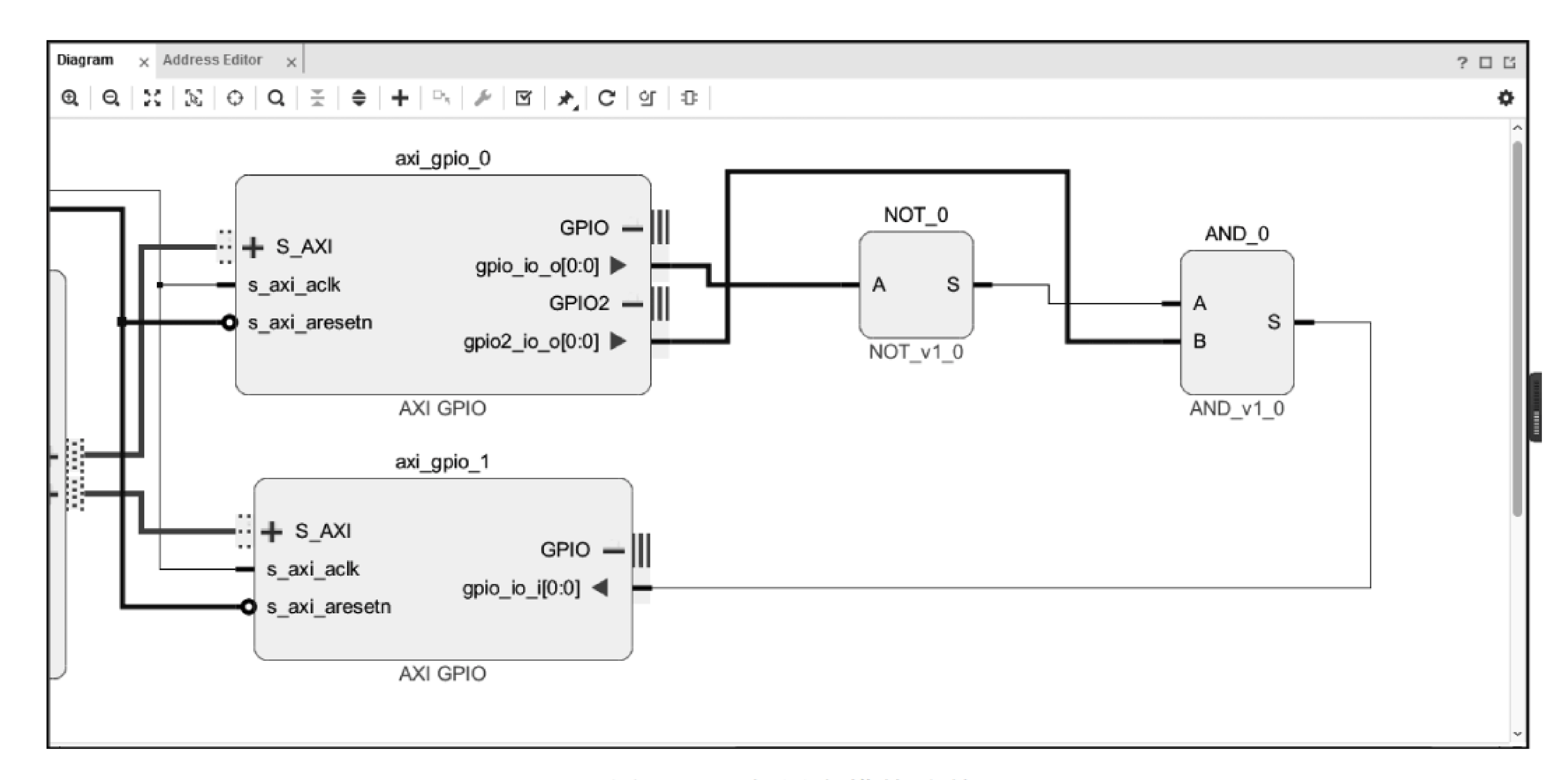


图 4-59　与用户模块连接

5. 综合

对刚才创建的 design_1.bd 进行 HDL 封装，并将封装后的 design_1_wrapper 设置为顶层文件。单击 project_2 主界面的 SYNTHESIS→run synthesis，会出现 Lauch Runs 弹出窗口，单击 OK 按钮进行综合。

6. 实现

当综合完成时会出现 Synthesis Completed 弹出窗口，选择 Run Implement 进行实现。

7. 生成比特流

当实现完成时会出现 Implement Completed 弹出窗口，选择 Generate Bitstream 来生成比特流。

选择 Files→Export→Export Block Design，如图 4-60 所示，选择存放路径，导出 design_1. tcl，如图 4-61 所示。

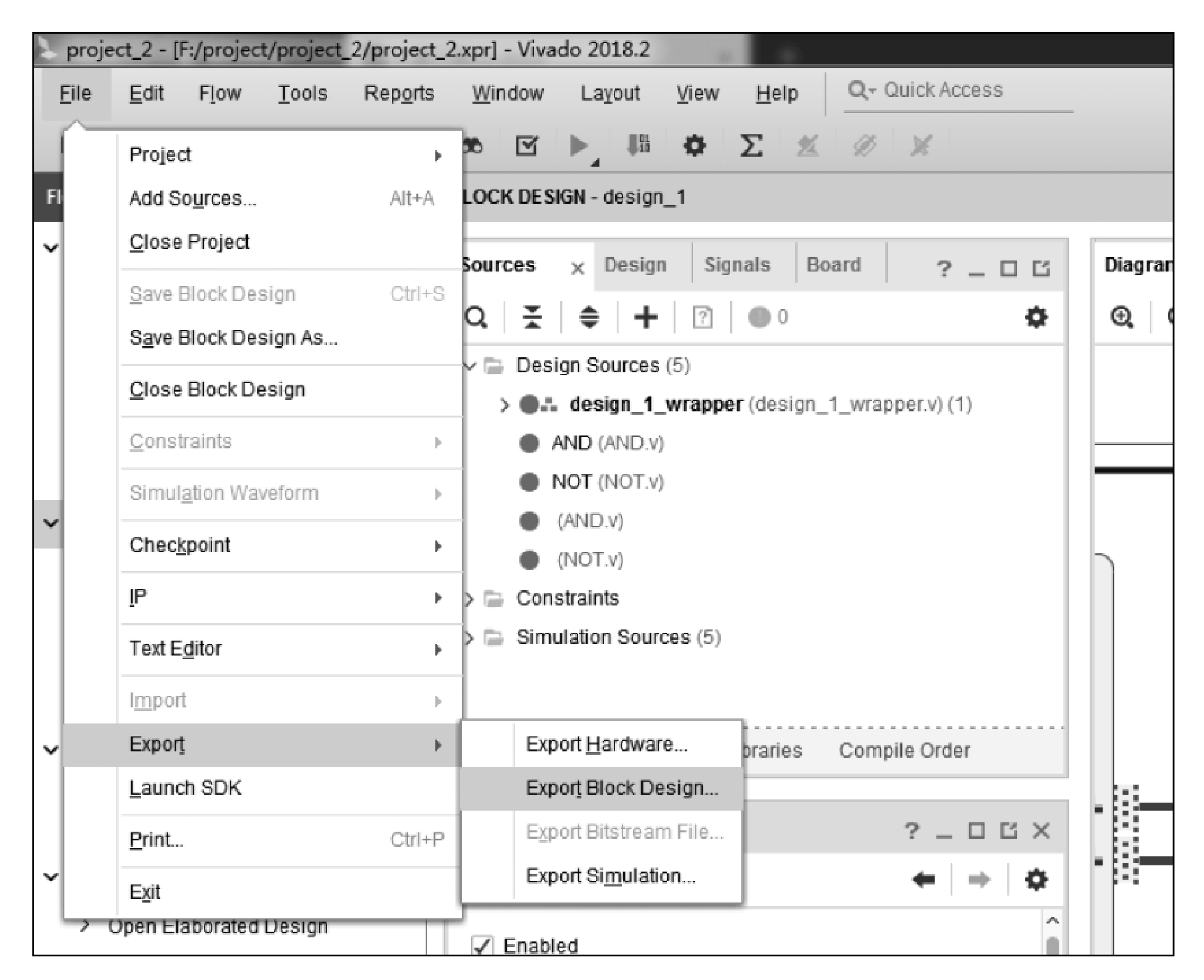


图 4-60　导出 TCL

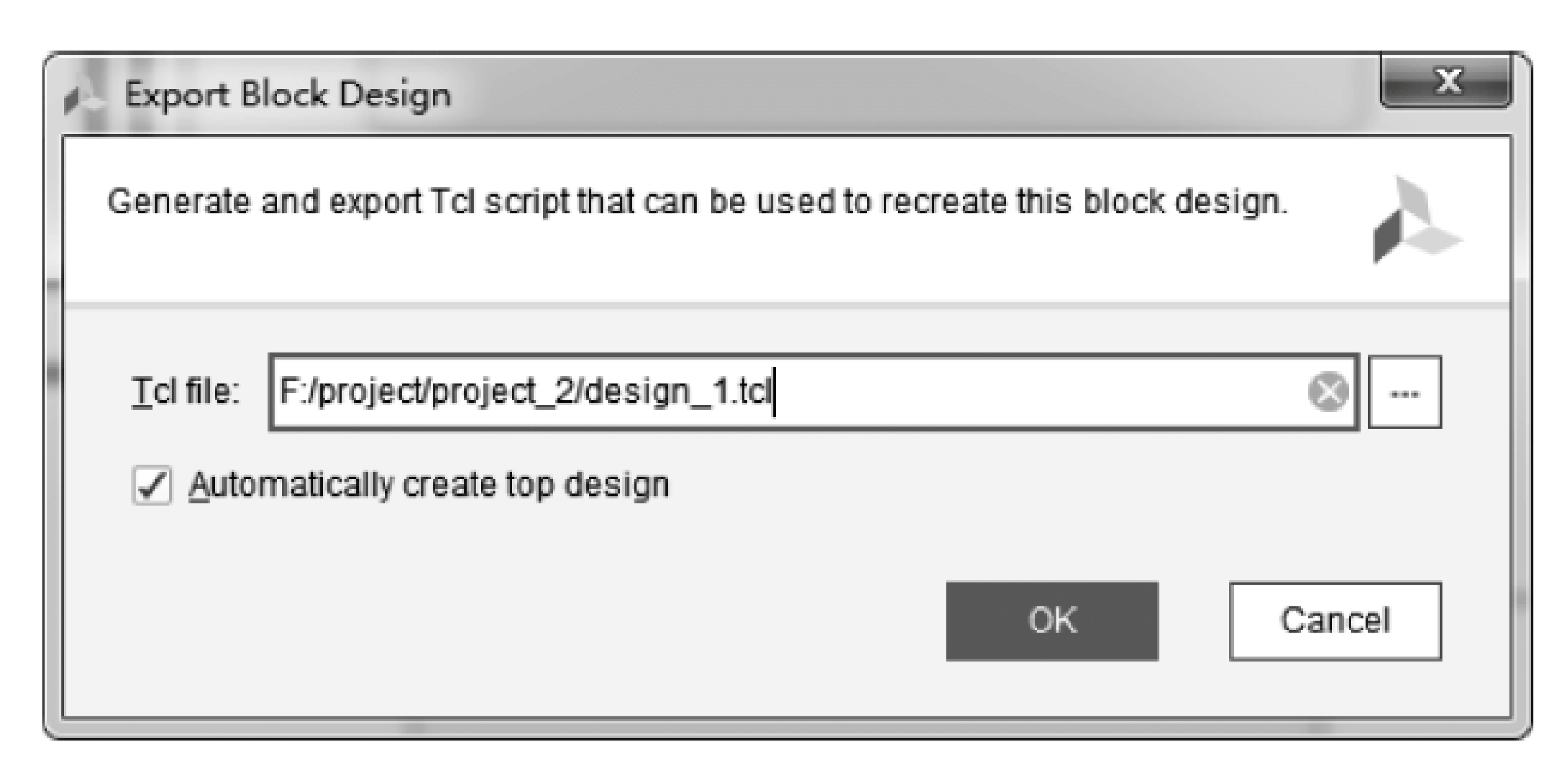


图 4-61　导出 TCL

在 project_2/project_2. runs/impl_1 目录下找到 design_1_wrapper. bit，并将其复制到 project_2 目录下，使其和 design_1. tcl 在同一目录下。再将 design_1_wrapper. bit 重命名为 design_1. bit，使其和. tcl 文件同名，如图 4-62 所示。

4.8.2　Verilog 方式

Verilog 方式包括创建工程、设计输入、仿真、与 PS 连接、综合、实现、生成比特流等。



图 4-62　bit 流文件

1. 创建工程

参照 4.1 节在 boards 列表中选择 PYNQ_Z2，完成工程(project_3)的创建。

2. 设计输入

在工程创建完成之后，右击，选择 Design Source 选择 Add Sources 添加或创建设计文件，再选择 Create File，创建名为 example.v 的 Verilog HDL 源文件。

代码如下：

```
module example(
    input A,
    input B,
    output S
    );
    assign S = (~A)&B;
Endmodule
```

3. 仿真

在主界面右击，选择 Simulation Source，选择 Add Source，然后选择 Add or create simulation sources，再选择 Create File，创建名为 test.v 的仿真文件。

Testbench 示例如下：

```
module test( );                     //Testbench 模块名，为顶层模块，且没有输入/输出
    reg A;
    reg B;
    wire S;
    example u1(                     //例化 example.v
        .A_0(A),
        .B_0(B),
        .S_0(S)
        );
    initial
    begin
        #20                         //延时 20 个时间单位
        A = 1'b1;
        B = 1'b0;
        #20                         //延时 20 个时间单位
```

```
        A = 1'b0;
        B = 1'b0;
        #20                    //延时 20 个时间单位
        A = 1'b0;
        B = 1'b1;
        #20                    //延时 20 个时间单位
        A = 1'b1;
        B = 1'b1;
    end
endmodule
```

Testbench 编写完成后,在左侧工程管理栏中选择 Simulation→Run Simulation→run behavioral simulation 可进行行为仿真验证,仿真结果与原理图方式一样。

4. 与 PS 连接

单击 Tools,选择 Create and Package New IP,在 Packaging Options 中选择 Package your current project,将 example.v 打包成 IP。

单击工程界面左侧的 IP Integrator 下的 Creat Block Design,创建一个新的 Block Design。参照 4.5.2 节完成对 PS 端的配置,不同的是需要添加两个 AXI_GPIO 模块,如图 4-53 所示。删除两个 GPIO 的 arduino_a0_a5 接口。

双击 axi_gpio_0,在 Board 中将 Board Interface 设置为 custom,如图 4-54 所示。在 IP Configuration 中按图 4-55 进行设置,为 GPIO 设置两个 1 位输出端口。

双击 axi_gpio_1,在 Board 中将 Board Interface 设置为 custom。在 IP Configuration 中按图 4-56 进行设置,为 GPIO 设置一个 1 位输入端口。此时 Block Design 如图 4-57 所示。

添加之前生成的 example_v1_0 这个 IP,并与两个 GPIO 按图 4-63 进行连接,完成 PS 和 PL 的连接。

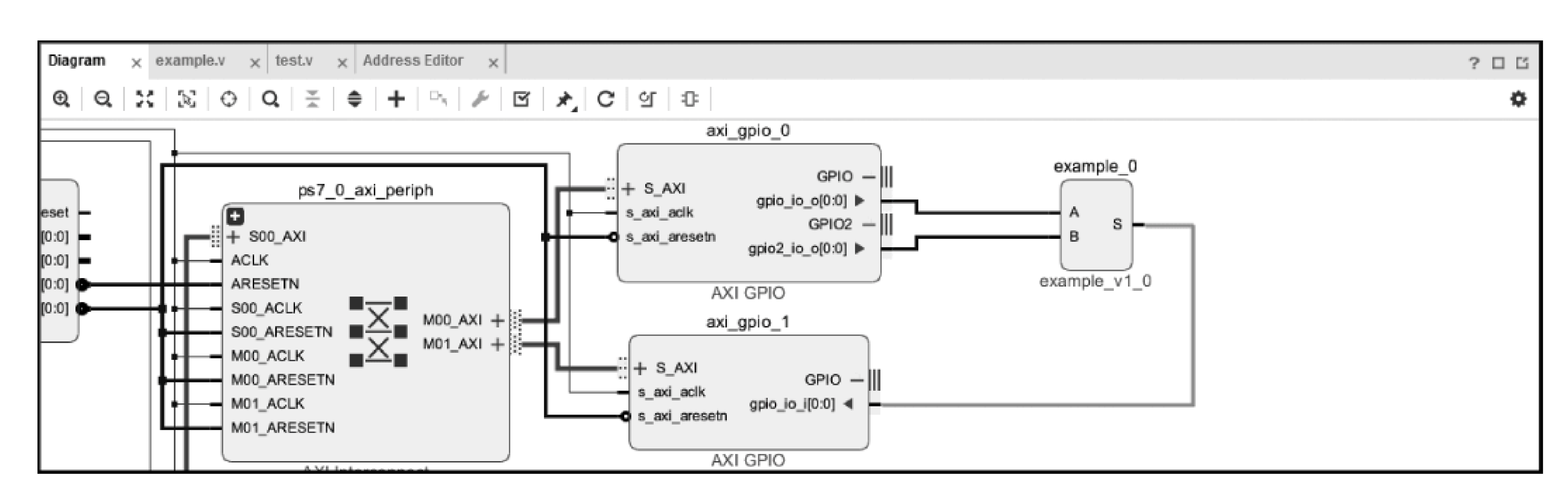


图 4-63 连接图

5. 综合

对刚才创建的 design_1.bd 进行 HDL 封装,并将封装后的 design_1_wrapper 设置为顶层文件。单击 project_3 主界面的 SYNTHESIS→run synthesis,出现 Lauch Runs 弹出窗口,单击 OK 按钮进行综合。

6. 实现

当综合完成时会出现 Synthesis Completed 弹出窗口,选择 run Implement 进行实现。

7. 生成比特流

当实现完成后会出现 Implement Completed 弹出窗口,选择 Generate Bitstream 来生成比特流。

选择 Files→Export→Export Block Design,出现如图 4-64 所示界面,再选择存放路径,导出 design_1. tcl,如图 4-65 所示。

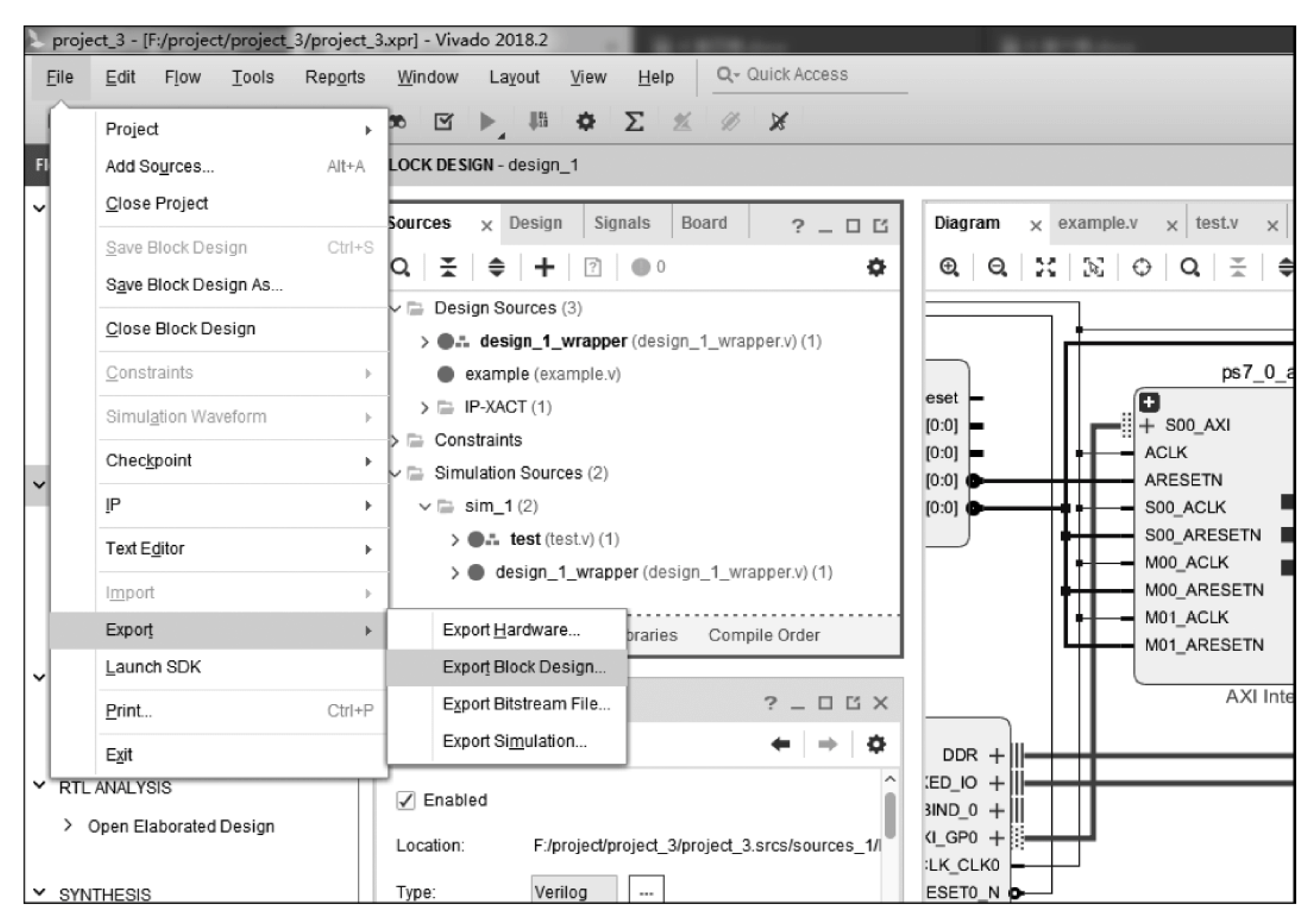


图 4-64 导出 TCL(一)

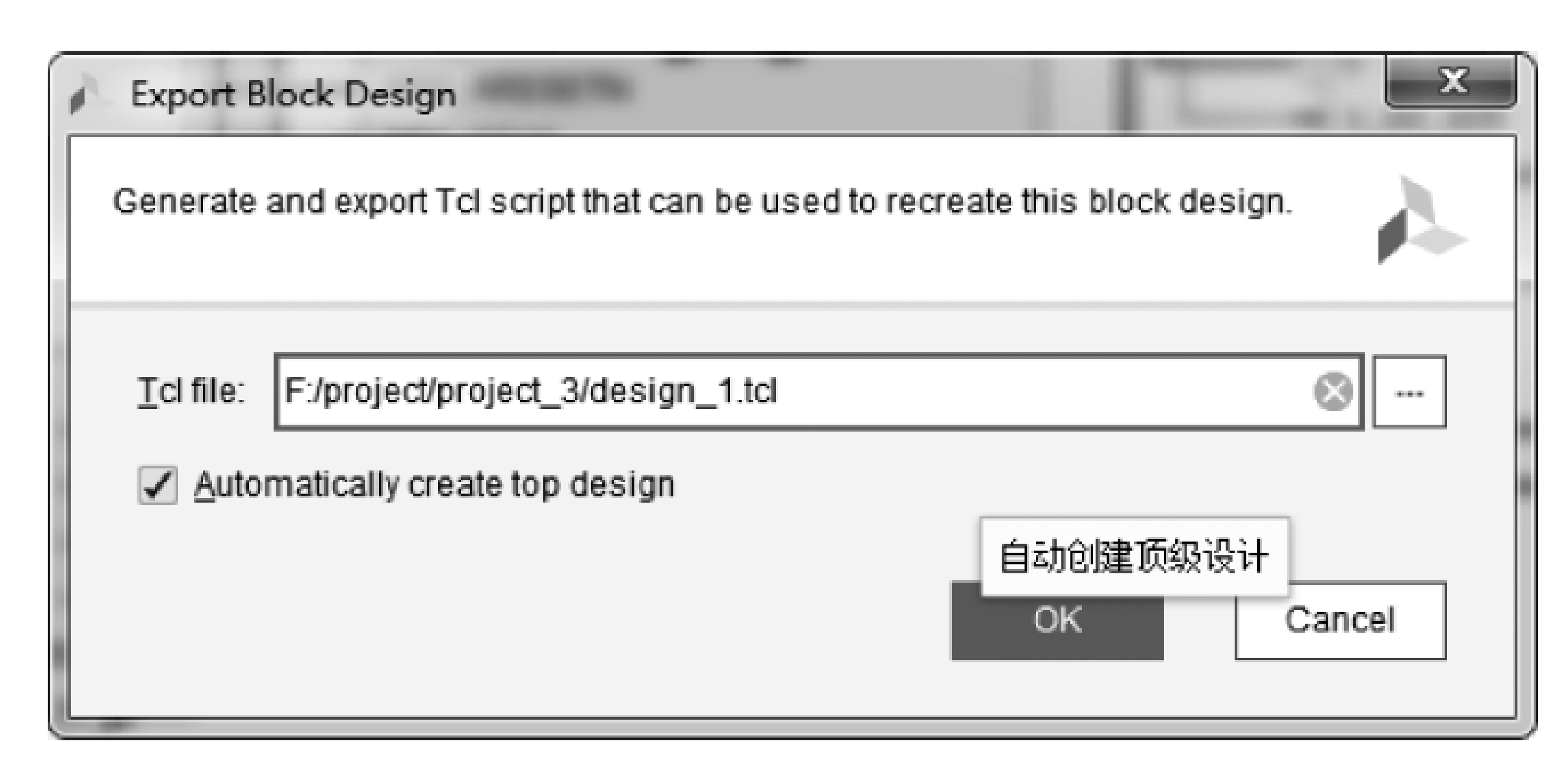


图 4-65 导出 TCL(二)

在 project_3/project_3. runs/impl_1 目录下找到 design_1_wrapper. bit,并将其复制到 project_3 目录下,使其和 design_1. tcl 在同一目录下,再将 design_1_wrapper. bit 重新命名

为 design_1. bit，使其和. tcl 文件同名，如图 4-66 所示。

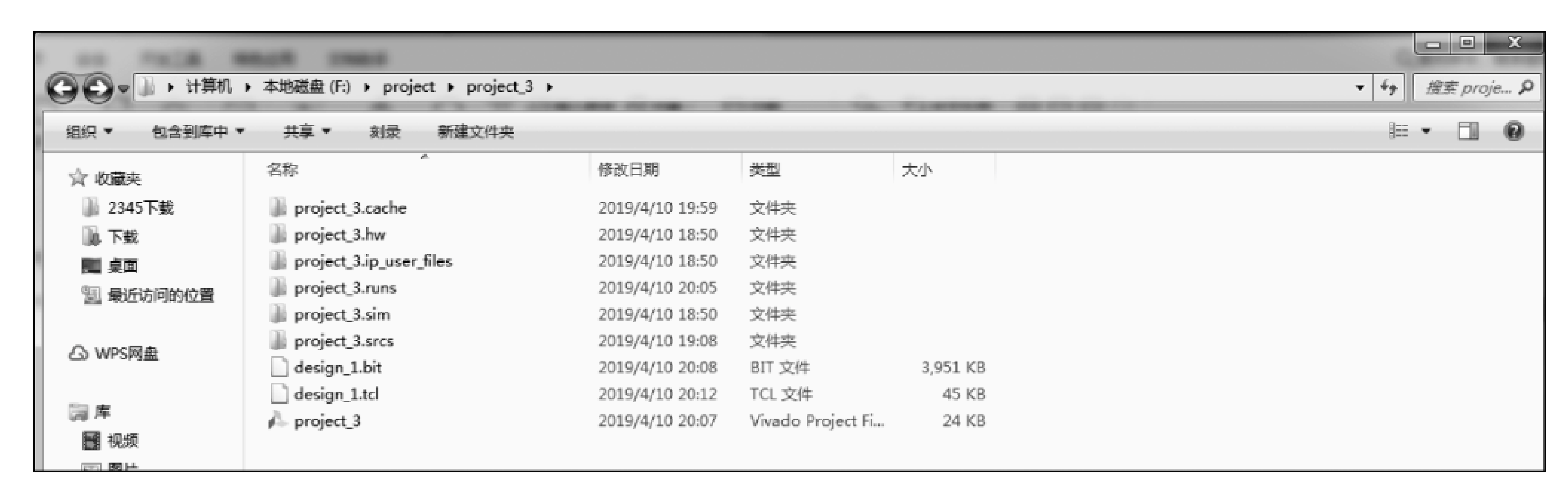


图 4-66　比特流文件

第 5 章 基于 Python 的 I/O 交互

CHAPTER 5

前面章节介绍了如何使用 Vivado 设计自己的硬件模块，以及如何通过 AxiGPIO 将自己设计的模块与 PS 连接起来。本章主要介绍基于 Jupyter Notebook，通过 Python 代码的方式操作 AxiGPIO 与用户设计的硬件模块进行交互。

5.1 Jupyter Notebook 介绍

Jupyter Notebook 是 Fernando Perez 发起的 IPython 项目。IPython 是一种交互式 shell，与普通的 Python shell 相似却具有一些很好的功能，如语法高亮显示和代码补全等。Jupyter Notebook 的工作方式是通过 Web 浏览器打开并编辑 Python 源代码，将其消息发送给 IPython 内核，由内核执行代码，然后将结果发送回浏览器的 Notebook。IPython 内核是在后台运行的 IPython 应用程序，本书介绍的 IPython 内核是运行在 Zynq-7000 的 PS 上的。

Jupyter 的核心是 Notebook 服务器，通过浏览器连接到该服务器。而 Notebook 呈现为 Web 应用。保存 Notebook 时，它将作为 JSON 文件（文件扩展名为.ipynb）写入到该服务器中。

采用 Jupyter Notebook 架构的一个优点是：内核无须运行 Python。由于 Notebook 和内核相互独立，因此可以在两者之间发送任何语言的代码。例如，早期的两个非 Python 内核分别是 R 语言和 Julia 语言。使用 R 内核时，用 R 编写的代码将发送给执行该代码的 R 内核，这与在 Python 内核上运行 Python 代码完全一样。IPython Notebook 如今已被更名为 Jupyter Notebook，因为 Notebook 变得与编程语言无关。新的名称 Jupyter 由 Julia、Python 和 R 组合而成。

第二个优点是可以在任何地方运行 Notebook 服务器，并且可通过互联网访问服务器。通常会在存储所有数据和 Notebook 文件的自有计算机上运行服务器。但是，也可以在远程计算机或云实例（如本书介绍的 PYNQ 云节点）上设置服务器。之后，就可以在世界上任何地方通过浏览器访问 Notebook。

第三个优点是由于它们比 IDE 平台更具交互性，因此它们被广泛地应用于教学场景。

5.1.1 Jupyter 组件

Jupyter 组件的内容：

（1）Web 浏览器：交互式 Web 应用程序，用于交互式编写和运行代码以及编写笔记本文档。

（2）内核：由 Notebook Web 应用程序启动的独立进程，它以给定语言运行用户代码并将输出返回给 Notebook Web 应用程序。对于 PYNQ 来说，它是用 Python 编写的，是 Jupyter Notebook 的默认内核，也是 PYNQ 发行版中为 Jupyter Notebook 安装的唯一内核。

（3）Notebook 文档：包含 Notebook Web 应用程序中所有内容表示的自包含文档。

5.1.2　Notebook 基础

Notebook 服务器在 PYNQ 上 PS 端 ARM 处理器上运行。当本地电脑以任何方式与 PYNQ 开发板连接时，使用任意浏览器访问 pynq：9090 即可连接到 Jupyter Notebook。图 5-1 为 Jupyter Notebook 界面。

（1）要创建新的 Notebook 文档，请单击列表顶部的 New 按钮，然后从下拉列表中选择，如图 5-2 所示。

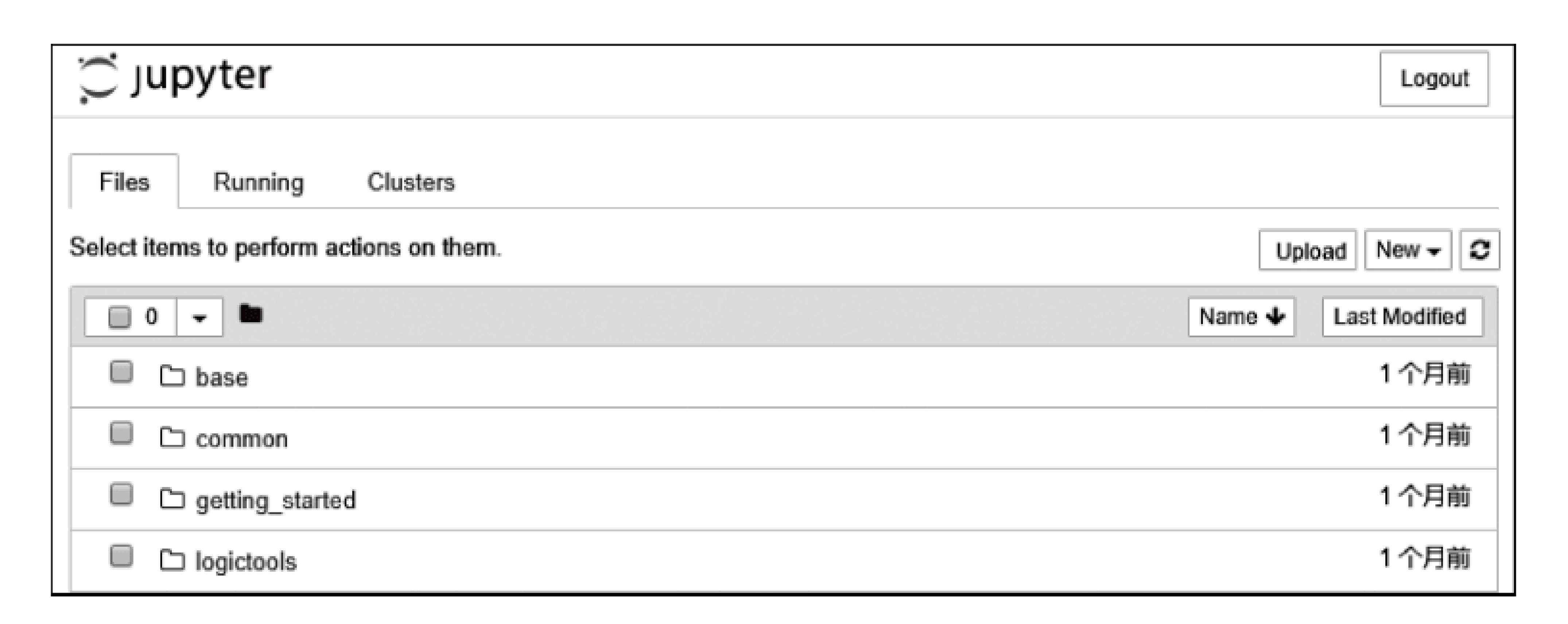


图 5-1　Jupyter Notebook 界面



图 5-2　新建文档

（2）列表中图标为绿色，且显示绿色 Running 文本的文件是正在运行的文件，如图 5-3 所示。直到明确关闭为止，这些文件会一直运行。

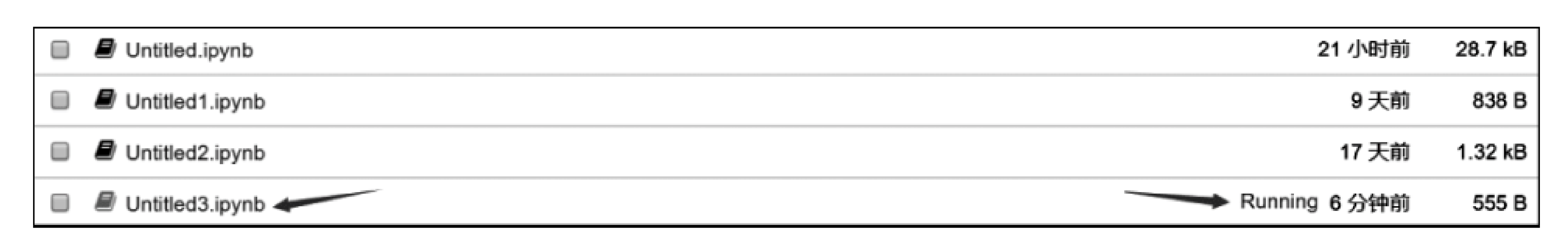


图 5-3　正在运行的文件

（3）要查看所有正在运行的文件及其目录，可以选择 Running 选项，如图 5-4 所示。

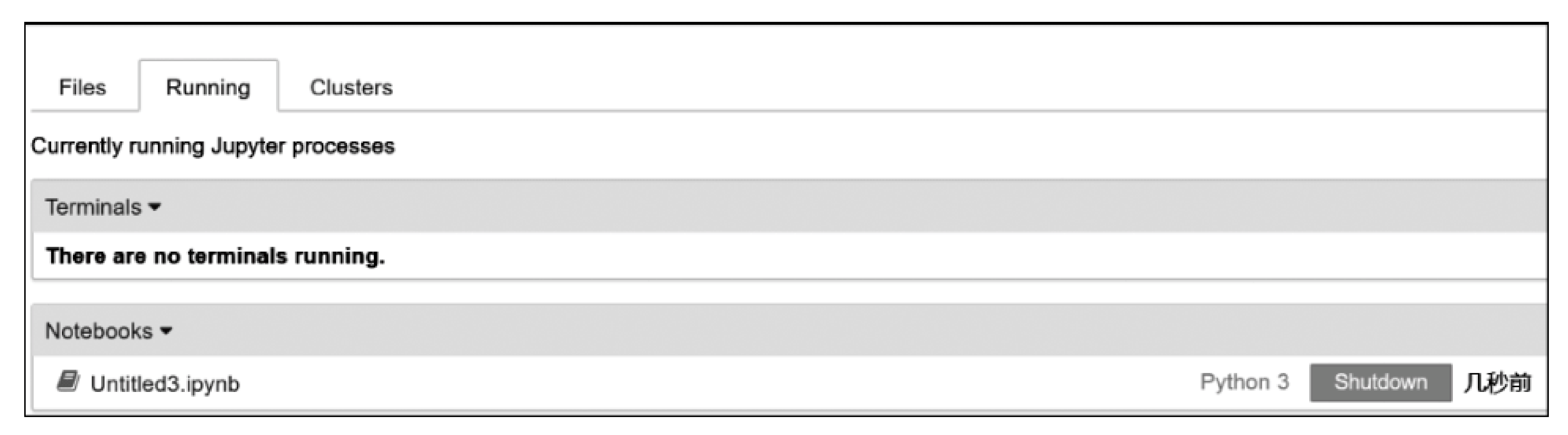


图 5-4　查看运行文件

5.1.3 Notebook 用户界面

创建文件或打开现有文件后，将进入 Notebook 用户界面。Notebook 用户界面具有以下主要区域：

(1) Notebook 的名称。

(2) 主工具栏提供了保存、导出、重载 Notebook，以及重启内核等选项。

(3) 快捷键。

(4) Notebook 主要区域，包含了 Notebook 的内容编辑区，界面如图 5-5 所示。

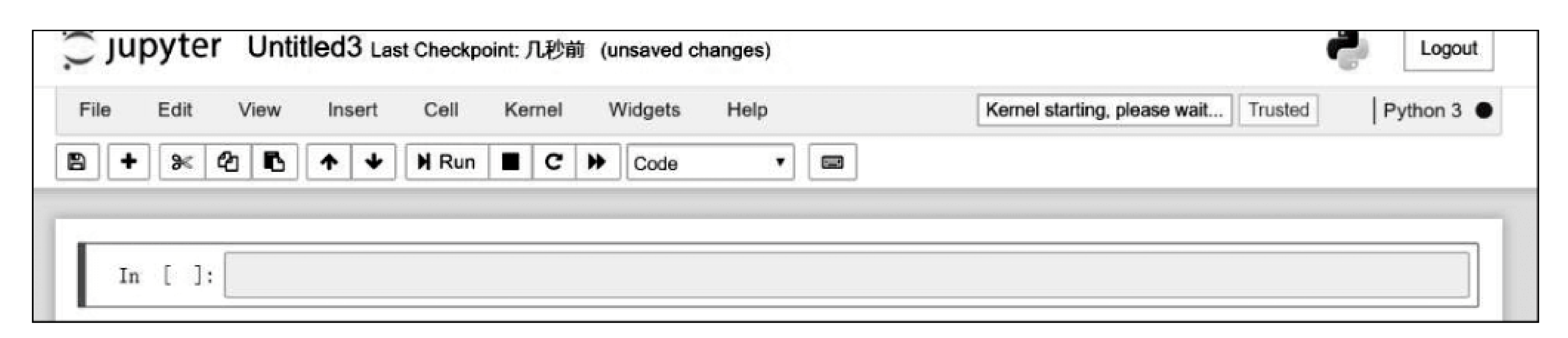


图 5-5 用户界面

Jupyter Notebook 有两种模式：编辑模式和命令模式。

1. 编辑模式

编辑模式由绿色单元格边框指示。当单元格处于编辑模式时，可以像普通文本编辑器一样输入单元格。按 Enter 键或单击单元格的编辑器区域可进入编辑模式，如图 5-6 所示。

```
In [ ]: def study():
            study=1
            print(study)
```

图 5-6 编辑模式

2. 命令模式

命令模式由带有蓝色左边距的灰色单元格边框表示，如图 5-7 所示。

```
In [1]: def study():
            study=1
            print(study)
        study()

        1
```

图 5-7 命令模式

Notebook 可以通过修改之前的单元格，对其重新运算，同时更新整个文档。例如将图 5-7 中的 print(study)改为 print("love")，然后按下 Shift+Enter 组合键重新计算该单元格，输出结果马上就更新成了"love"。通过这种方法可以对某一模块进行多次修改，可大大减少程序开发与调试的时间。

Notebook 中还有 Header 单元格和 Markdown 单元格两种不同的单元格，可以利用它们对代码进行美化，如图 5-8 所示。

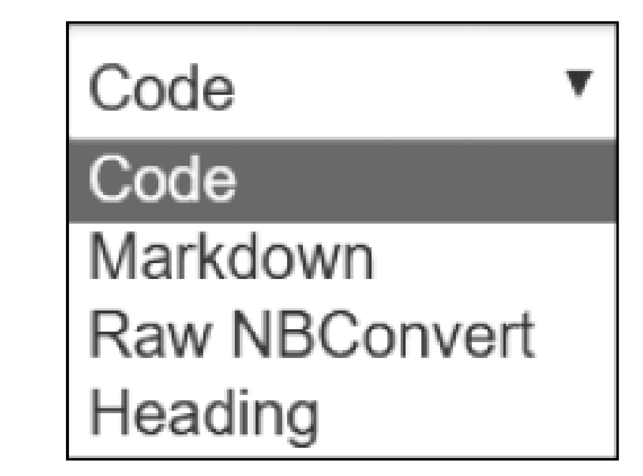


图 5-8 其他单元格类型

Heading 可以创建不同层级的标题，并将之与代码相区分，

使得代码的层次更加清晰。Markdown 可以对解释文本进行漂亮的渲染，产生美观的注释，效果如图 5-9 所示。

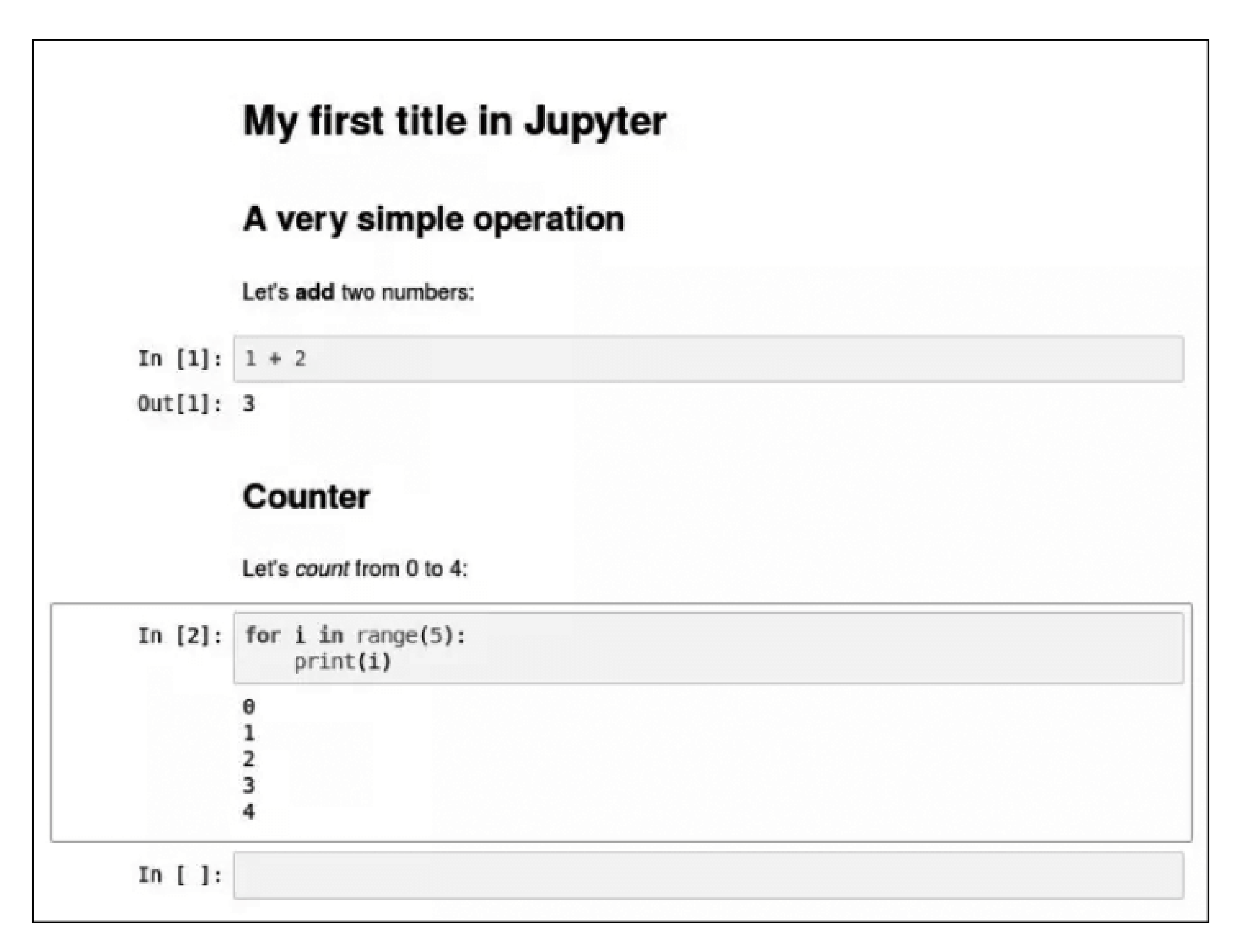


图 5-9　Heading 和 Markdown 使用效果

5.2　使用 PYNQ Overlay 加载流文件

用户设计的比特流文件可以通过 Vivado 工具下载到 PL 中，也可以在 Jupyter Notebook 中直接使用 Python 代码进行下载。PYNQ 中已经内置了 Jupyter Notebook。



图 5-10　上传文件

首先将 Vivado 中生成的 *.bit 和 *.tcl 这两个文件通过 Jupyter Notebook 导入到 PYNQ 板卡上。如图 5-10 所示，在 Jupyter Notebook 中单击 Upload，选择文件路径将上述两个文件加载进 Jupyter Notebook 可访问的文件夹中。

在使用 Python 代码下载比特流文件时，首先要引入 PYNQ 包中的 Overlay 文件，然后通过 Overlay 来加载比特流到 PYNQ 中。如示例中将 base.bit 流文件加载到 PYNQ 板卡上的 FPGA 芯片中去。想要配置自己的比特流文件，只需要生成命名相同的 *.bit 与 *.tcl 文件。例如：

```
from pynq import Overlay
overlay = Overlay("base.bit")    #加载流文件"base.bit"。Tcl 文件也是需要的,会进行解析
```

5.3　Python 引脚绑定

AxiGPIO 接口模块提供了从外部通用外设(如 led、按钮、通过 AxiGPIO 控制器 IP 连接到 PL 的开关)读取、写入和接收中断的方法。AxiGPIO 模块控制 PL 中 AxiGPIO 控制

器的实例。AxiGPIO 框图如图 5-11 所示，每个 AxiGPIO 最多有两个通道，每个通道位宽最大为 32 位。

通过 Python 调用 read()和 write()用于在通道上读写数据。

setdirection()和 setlength()可用于配置 IP。

AxiGPIO 的方向可以是 in、out 和 inout；AxiGPIO 默认的方向是 inout。设定了 in 或 out 后只允许对 IP 分别进行读和写操作，如果读取'out '或写入'in'将会报错。

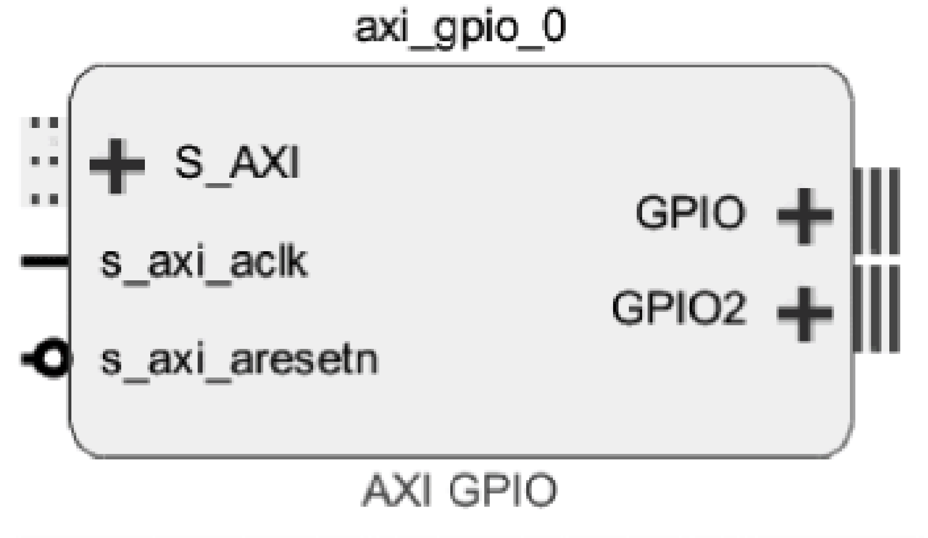


图 5-11　AxiGPIO 框图

官方文档中包含的例子介绍了如何使用 AxiGPIO 模块。在实践中，可以使用 LED、按钮、开关和 RGBLED 等来扩展 AxiGPIO，这些外围设备需要使用 Overlay 来加载 bit 文件。在加载 Overlay 之后，通过将 AxiGPIO 控制器的名称传递给实例化的 AxiGPIO 实例。

加载 bit 文件和端口分配的示例如下：

```
from pynq import Overlay
from pynq.lib import AxiGPIO
ol = Overlay("base.bit")

led_ip = ol.ip_dict['gpio_leds']
switches_ip = ol.ip_dict['gpio_switches']
leds = AxiGPIO(led_ip).channel1
switches = AxiGPIO(switches_ip).channel1
```

首先将 bit 文件通过 Overlay 函数赋值给变量 ol，再通过 led_ip = ol.ip_dict['gpio_leds']将采用 Vivado 设计时命名为'gpio_leds'的 AxiGPIO 模块赋值到 Jupyter Notebook 中的 led_ip 端口，完成 Jupyter 与 AxiGPIO 的连接。使用 leds = AxiGPIO(led_ip).channel1 语句对 AxiGPIO 的通道 1 进行定义。

简单的读写操作的示例如下：

```
mask = 0xffffffff
leds.write(0xf, mask)
switches.read()
```

AxiGPIO 需要包含掩码进行写操作。首先定义一组掩码值，再对定义完成的端口通过 write()和 read()进行数据的读写操作。

5.4　基于 Python 调试组合逻辑

通过 Python 调试组合逻辑方法相对比较简单，本书涉及的组合逻辑实验基本都使用相同的调试文件。文件主要使用 read()和 write()两个模块。下面的例子来自第 8 章加法器实验的 Python 程序。

编辑 Python 代码时需要调用很多的库函数，在 PYNQ 平台上内置了很多的库函数，使用时可以直接调用。

```
from pynq import Overlay                    # 调用 Overlay 库
from pynq.lib import AxiGPIO                # 调用 AxiGPIO 库
overlay = Overlay("/home/xilinx/jupyter_notebooks/… .bit")    # bit 文件路径
overlay?                                    # 显示出 bit 文件中的各种 IP 信息
```

对各个端口进行命名：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']    # 将使用的 AxiGPIO 模块 Jupyter Notebook 中进行命名
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_2 = overlay.ip_dict['axi_gpio_2']

a = AxiGPIO(gpio_0).channel1              # 对 AxiGPIO 上的各个通道进行命名
b = AxiGPIO(gpio_0).channel2
cin = AxiGPIO(gpio_1).channel1
s = AxiGPIO(gpio_1).channel2
cout = AxiGPIO(gpio_2).channel1
```

对 AxiGPIO 的输出端口使用 write()进行赋值，将数据送到 PL 端进行处理。待 PL 端处理结束后将结果送回 AxiGPIO 的输入端，并通过 read()读取出处理后的结果。通过 AxiGPIO 输出数据时需要掩码才能输出，一般默认掩码每个位都为 1。Python 中 4 位十六进制的掩码为 0xf，用户可以根据输出数据的位数改变掩码的位数。如第 4 章介绍，Channel 的位宽是用户在 Vivado 中调用 AxiGPIO 时设置好的。

```
mask = 0xf                                # AxiGPIO 掩码
a.write(8,mask)                           # 将 AxiGPIO 的输出端口赋值为 8
b.write(9,mask)
cin.write(0,mask)
```

GPIO 接收端得到数据后不会显示出来，此时需要使用 print()函数来显示得到的结果。

```
print(cout.read())                         # 将 AxiGPIO 接收端得到的结果显示出来
print(s.read())
```

5.5　基于 Python 调试时序逻辑

时序逻辑电路通过 Python 进行交互的时候，由于 PL 端并行度高，数据处理速度快，而 PYNQ 平台上的 AxiGPIO 模块有延时。当数据处理量较大时，有时会造成 AxiGPIO 读取时序电路数据时出现数据遗漏，造成数据不完整。为了解决这一问题，将时序电路的时钟输入端口通过 Python 调试文件进行输出，保证 Python 调试程序的时钟与 PL 端相一致。让每个时钟周期的数据都能被记录下来。下面使用包含时序电路的计数器 Python 调试模块进行介绍。

读取比特文件和对各端口进行命名这两部分与逻辑电路时基本相同，在此不再赘述。

```
from pynq import Overlay
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/…bit")
overlay?

gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_2 = overlay.ip_dict['axi_gpio_2']

clk = AxiGPIO(gpio_0).channel1              #对时钟所在通道进行命名
rst_n = AxiGPIO(gpio_0).channel2            #对复位所在通道进行命名
out0 = AxiGPIO(gpio_1).channel1
```

在时序逻辑中引入时间函数来进行延时，保证数据有足够的时间写入读出，防止出现错误。复位端口低电平有效，不复位时该端口要置 1。

```
import time                                 #调用时间函数
mask = 0b1                                  #AxiGPIO 掩码
timedelay = 0.1                             #设置时间延时
rst_n.write(0b0,mask)                       #进行复位
rst_n.write(0b1,mask)
```

时钟的变化为高低电平交替，可通过 Python 调试文件设置时钟的变换，并通过 For 循环来模仿时钟连续性。

Python 中 for 循环包含三个变量，第一位为循环的起始位，第二位为循环的次数，第三位为循环步长。当步长为 1 时每循环一次循环次数加 1，如果步长为 N，则每次循环后循环次数加 N。

由于每个时钟周期后的数值都需要被记录，直接输出结果观察起来会很不方便，通过使用数组可以保证每个时钟的结果都单独保存。每次循环中都建立一个空白的数组，并通过 append 函数将数值添加到数组中保存起来。

```
import time
N = 30
timedelay = 0.1
sizes = range(0,N,1)                        #设置循环次数
mask = 0b1
for size in sizes:
clk.write(0b0,mask)                         #创造时钟
clk.write(0b1,mask)
tmp = []                                    #创建数组保存结果
a1 = out0.read()
tmp.append(a1)
print(tmp)                                  #打印数据，读出每个时钟中周期的结果
```

5.6　实例演示

在参照 4.8 节生成. bit 和. tcl 文件后，请查看第 2 章完成 PYNQ 实验环境的准备，之后按照以下步骤完成基于 PYNQ 的 Python 调试。

5.6.1　上传. bit 和. tcl 文件

通过选择 New→Folder 新建一个文件夹，选中这个文件夹将其重命名为 Example。

将 4.8 节生成的 design_1. bit 和 design_1. tcl 上传到 Jupyter Notebook 新建的 Example 文件夹下，如图 5-12 所示。

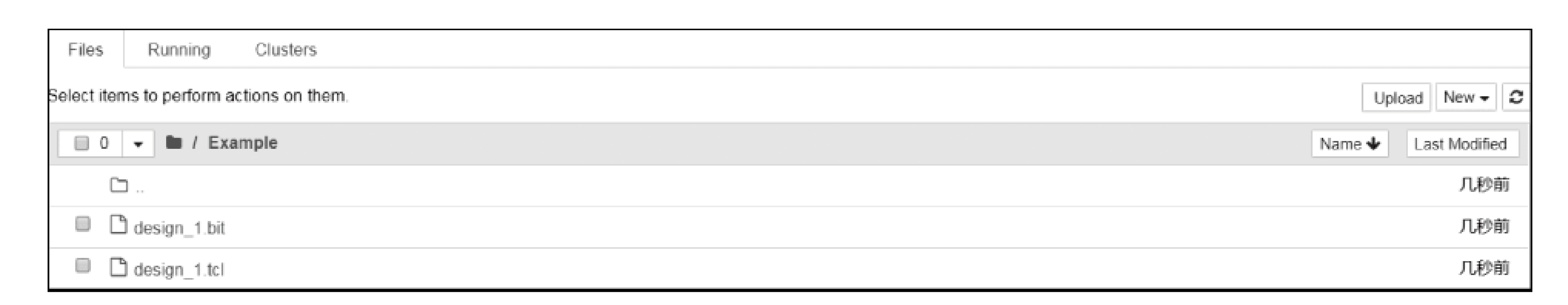


图 5-12　上传相关文件

5.6.2　基于 Python 的 I/O 交互

基于 Python 的 I/O 交互过程如下：

1. 创建 Notebook

首先通过选择 New→Python 3 创建一个 Notebook，如图 5-13 所示。

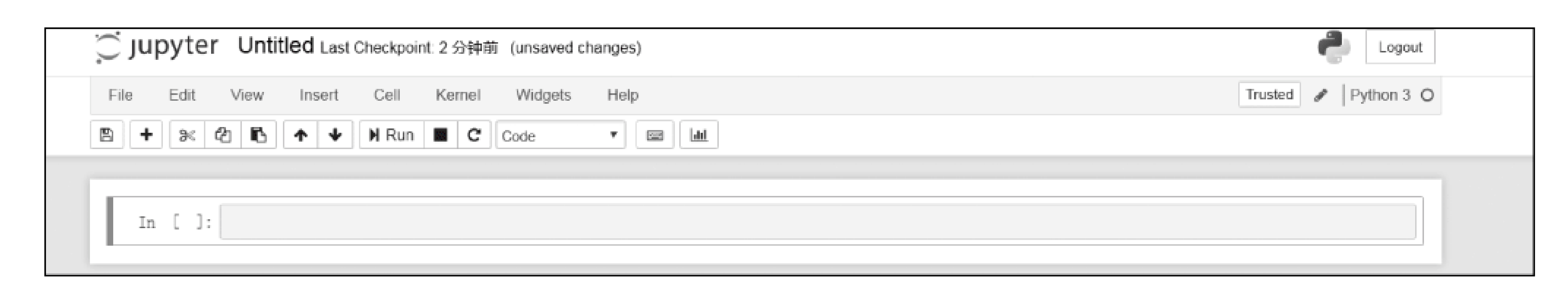


图 5-13　创建 Python 文件

2. 使用 pynq Overlay 加载比特流

代码如下：

```
from pynq import Overlay
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/Example/design_1.bit")
overlay?
```

通过“Overlay?”可以查看 overlay 的信息，在这里可以看到设计中使用到的两个 AxiGPIO，如图 5-14 所示。

3. 对端口的访问

对于端口的访问要和原理图连接一致，代码如下：

```
Type:           Overlay
String form:    <pynq.overlay.Overlay object at 0xb431cad0>
File:           /usr/local/lib/python3.6/dist-packages/pynq/overlay.py
Docstring:
Default documentation for overlay /home/xilinx/jupyter_notebooks/Example/design_1.bit. The following
attributes are available on this overlay:

IP Blocks
----------
axi_gpio_0          : pynq.lib.axigpio.AxiGPIO
axi_gpio_1          : pynq.lib.axigpio.AxiGPIO
```

图 5-14　Overlay 信息

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_A = AxiGPIO(gpio_0).channel1
gpio_B = AxiGPIO(gpio_0).channel2
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_S = AxiGPIO(gpio_1).channel1
```

4. GPIO 口的读写

对 gpio_A 和 gpio_B 两个端口写入 0 和 1，通过读取 gpio_S 值来判断是否与预期的结果（$S=\overline{A}B$）一致，进而判断设计是否正确。代码如下：

```
mask = 0b1
gpio_A.write(0b0,mask)
gpio_B.write(0b1,mask)
S = gpio_S.read()
print(S)
```

第6章 硬件描述语言简介

CHAPTER 6

硬件描述语言(Hardware Description Language,HDL)是以编写代码的方式来描述硬件模块及其功能的语言。相比原理图,硬件描述语言在描述大型复杂硬件系统时效率更高。常用的硬件描述语言有 Verilog HDL 和 VHDL。Verilog HDL 的优点是语法类 C,由于计算机类的学生通常具有 C 语言的编程基础,因此可以快速掌握 Verilog HDL 的编写。但缺点也是因为语法太像 C 语言,学生容易与软件编程混淆,从而用软件设计的思维来进行 Verilog HDL 编写,最终导致不可综合的设计。而 VHDL 相反,其语法更加严谨并需要一定的重新学习过程,但优点是没有软件编程的历史包袱,一开始学生就以硬件的思维来使用它。因为硬件描述语言本质上是对某个硬件模块的描述,因此 Verilog HDL 和 VHDL 逻辑上是等价的。总体来说,使用任何一种语言都是可以的,在同一个工程中,这两种设计语言也可以混合使用,只要同一个模块选定一种就可以。企业在招聘时也通常将这两种语言的技能同等看待。

本章仅介绍数字逻辑及计算机组成原理实验所需要的最基本的 HDL 知识,方便学生快速入门。鉴于网络上有大量公开的 HDL 教程,更为详细的介绍请自行参考相关资料。考虑到不同学校的使用习惯,这里给出了 Verilog 和 VHDL 对照的版本。也希望更有利于读者理解 HDL 是硬件描述的本质。

6.1 "模块"的描述

"模块"是 Verilog HDL 和 VHDL 描述的最基本单元,复杂的系统则可以通过模块的层次化嵌套使用来完成。如图 6-1 是两种语言对模块的描述:在 VHDL 中,模块描述的关键词叫 Entity,也就是实体,Entity 部分只关注该模块的外在表现如模块名称、输入/输出端口。而对于模块内部功能的详细描述是由 Entity 对应的 Architecture 部分来完成的。Architecture 的描述也分为两部分:在 Architecture 和 Begin 之间是对该模块所用的变量和信号进行的声明。Begin 之后才是真正的模块功能的详细描述。

Verilog HDL 的语法则要简洁一些。Verilog 中模块的关键字就是 Module,而对于该模块的输入/输出端口、内部所用的变量以及模块功能的详细描述都在 Module 和 Endmodule 之间的区域内完成。

VHDL	Verilog HDL
entity 实体名 is	module 模块名(端口表)
port(输入/输出端口说明)	输入/输出端口说明
end	
Architecture …is	
内部信号变量说明;	内部说明部分;
begin	
并行执行语句;	并行执行语句;
end 结构体名;	endmodule

图 6-1 模块描述的基本框架

6.1.1 输入/输出端口说明

下面举例说明输入/输出端口的声明:

1. VHDL

```
Entity mux4 is
    Port (s: in std_logic_vector(1 downto 0);
    a,b,c,d: in std_logic_vector(7 downto 0);
    y: out std_logic_vector( 7 downto 0));
End mux4;
```

在 VHDL 中,模块输入/输出端口的声明是在 Entity 中,在此说明每个输入/输出端口的名称、方向、端口的类型及宽度。例如 s: in std_logic_vector(1 downto 0),其中 s 是端口名称; in 是端口的方向,说明此处是输入端口; std_logic_vector 是端口的类型,此处是标准的逻辑向量类型; (1 downto 0)是端口的宽度,此处是 2 位,最右边是第 0 位。

2. Verilog HDL

```
module kmux4_1(s, a, b, c, d, y);
    input[1:0] s;
    input [7:0] a, b, c, d;
    output[7:0] y;
```

在 Verilog 中,模块名称后面的括号里,可以只列出输入/输出端口的名称,如(s, a, b, c, d, y)。而在下面进一步说明端口的方向、数据宽度。如 input[1:0] s,input 是端口的方向,说明 s 是输入端口; 数据宽度是 2 位,最右边的是第 0 位。

6.1.2 数据对象和数据类型

VHDL 的数据对象有常量、信号和变量,分别表达不同的硬件对应概念。每种对象都可以设置为不同的数据类型,可以明确表达各种具体或抽象的数据。数据使用时必须进行类型说明,运算时必须考虑类型的一致性。

Verilog HDL 的数据对象有常量和变量。其中变量分为连线型(wire)和寄存器型

(reg)，wire 型类似于信号。而 reg 型类似于变量，只能在子程序块中赋值。数据变量默认为 wire 型。在 VHDL 中与之对应的是 signal(与 wire 对应)和 variable(与 reg 对应)。

6.1.3　顺序语句与并行语句

两种语言的语句都包含并行语句和顺序语句。并行语句在主程序中使用，其代码出现的顺序和执行的顺序无关，所有的并行语句是并行执行的。而顺序语句只能在子结构中使用，执行顺序和代码的出现顺序相关。

如图 6-2 所示，VHDL 中的 process…end process 之间以及 Verilog 中的 always…end 之间都是并行语句块。意思是多个 process，多个 always 代表的语句块是得到并行执行的，和其出现的顺序无关。而其内部的语句是顺序语句，执行是按出现的先后顺序进行的。

举例说明：

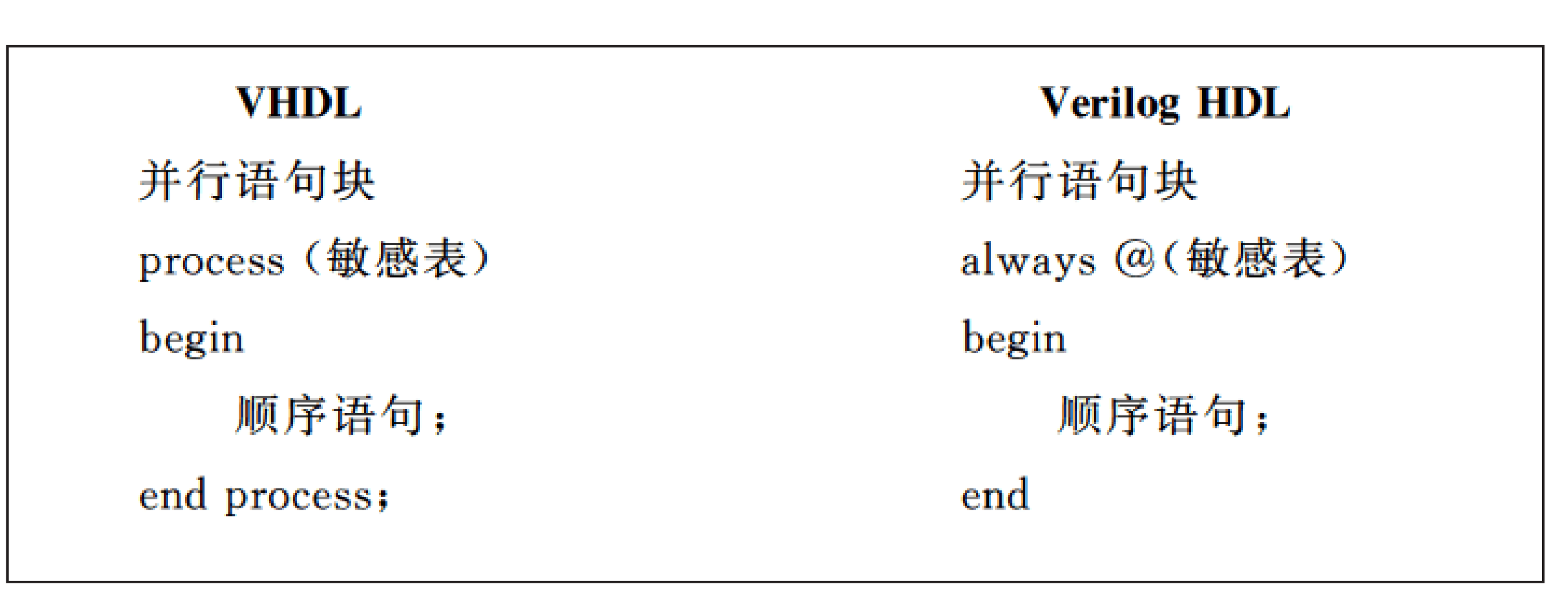


图 6-2　并行语句与串行语句

因此，如果想要使多个语句得到并行执行，就将其放入到多个 process/always 中(也许使隐含的，即不显式地出现 process/always 的关键字)；如果想要使多个语句得到先后的串行执行，就将其放入到一个 process/always 语句块的内部。

6.2　模块基本用法示例

下面用若干基本功能部件的 HDL 描述为例，说明模块的基本用法。

6.2.1　八位乘法器

用 VHDL 和 Verilog HDL 代码描述八位乘法器。

1. VHDL 描述

```
LIBRARY IEEE;                        --引入 IEEE 库
USE IEEE.STD_LOGIC_1164.ALL;         --引入 1164 包
ENTITY mul IS
port(    a, b: IN integer range 0 to 255;
         q: OUT integer range 0 to 65535);
END mul;
ARCHITECTURE one of mul is
BEGIN
    q<=a * b;                        --此处做乘法
END one;
```

2. Verilog HDL 描述

```
module mul8v (a, b, q);
    input[7:0]a, b;              //端口定义和乘法在同一个区域完成
    output[15:0] q;
    assign q = a * b;            //此处做乘法
endmodule
```

从上述的代码示例中可以更好地理解 HDL 是对硬件的描述,不同的 HDL 可以描述同一个硬件模块,只是具体语法稍有差异。

从上述例子中还可以体会 HDL 在描述硬件时的优点,就是当设计一个乘法器时,并不一定需要完全了解乘法器的内部架构的细节,而是仅用 a * b 表示。通过 EDA 工具综合之后生成乘法器的具体硬件架构。因此一定程度上提升了抽象层次,降低了对用户的底层硬件细节能力的需求。这种描述硬件行为而不是硬件具体架构细节的方式就叫作“行为描述”,与之对应的描述底层细节架构的方式叫作“结构描述”。在进行系统设计时通常采用的是上述两种描述方式的混合模式。复杂度高的模块先用结构描述拆分成多个子模块的连接,对于功能简单的模块再采用行为描述,这样设计的系统较方便成功综合。

6.2.2 译码器

三—八译码器的示意图如图 6-3 所示。ENA 是译码器的使能控制输入端,当 ENA=1 时,译码器不工作,此时 Y[7:0]=11111111(译码器的输出有效电平为低电平)。当 ENA=0 时,译码器工作。C、B、A 是 3 位宽度的数据输入端。译码器处于工作状态时,当 CBA=000,Y[7:0]=11111110(即 Y[0]=0);当 CBA=001,Y[7:0]=11111101(即 Y[1]=0);以此类推。

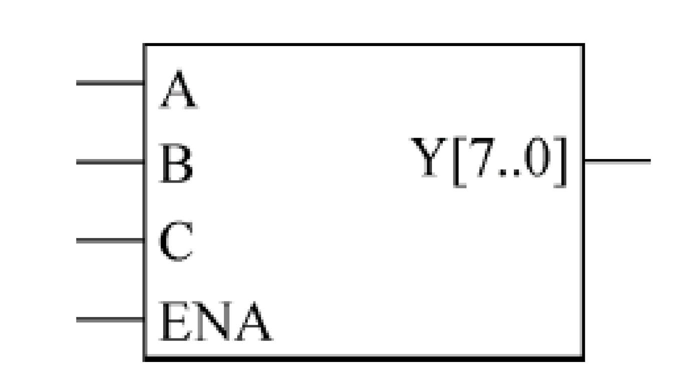


图 6-3 三—八译码器示意图

1. 三—八译码器的 VHDL 描述

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY Decoder IS
    PORT(a, b, c, ena: IN BIT;
        y: OUT BIT_VECTOR(7 DOWNTO 0));
END Decoder;
ARCHITECTURE one OF Decoder IS
BEGIN
  PROCESS(a, b, c, ena)          -- 这是一个可并行的语句块,a, b, c, ena 称为敏感信号,
                                 -- 当发生变化时,会激发 process 中的语句执行
    VARIABLE cba: BIT_VECTOR(2 DOWNTO 0);         -- cba 是变量只能在 process 内部声明
                                                  -- 和使用,受到激发后值立即发生变化
     BEGIN
        cba: = (c & b & a);                       -- 把 c、b、a 拼接起来变成 3 位的变量
        IF (ena = '1') THEN y <= "11111111";      -- 如果 ena 是 1,模块不工作
          ELSE
            CASE (cba) IS
```

```
                WHEN "000"  => y <= "11111110";          -- 下面这几条语句顺序执行
                WHEN "001"  => y <= "11111101";
                WHEN "010"  => y <= "11111011";
                WHEN "011"  => y <= "11110111";
                WHEN "100"  => y <= "11101111";
                WHEN "101"  => y <= "11011111";
                WHEN "110"  => y <= "10111111";
                WHEN "111"  => y <= "01111111";
                WHEN OTHERS => NULL;                     -- 如果不是上述值,则输出 NULL
            END CASE;
        END IF;
    END PROCESS;
END one;
```

2. 三—八译码器的 Verilog HDL 描述的代码示例

```
module  decoder(a, b, c, ena, y);
input     a, b, c, ena;
output[7:0]   y;
reg[7:0]     y;
always
begin
    if (ena == 1)   y  =  'b11111111;
    else begin
        case ({c,b,a})                          -- 把 c、b、a 用{}拼接起来形成 3 位
            'b000: y =  'b11111110;
            'b001: y =  'b11111101;
            'b010: y =  'b11111011;
            'b011: y =  'b11110111;
            'b100: y =  'b11101111;
            'b101: y =  'b11011111;
            'b110: y =  'b10111111;
            'b111: y =  'b01111111;
            default : y =  'b11111111;
        endcase
    end
end
endmodule
```

6.2.3　八位二进制比较器

设计一个八位二进制数据比较器，A[7:0]和 B[7:0]是两个数据输入端，FA 是“大于”输出端，FB 是“小于”输出端，FE 是“等于”输出端。当 A[7:0]大于 B[7:0]时，FA=1；当 A[7:0]小于 B[7:0]时，FB=1；当 A[7:0]等于 B[7:0]时，FE=1。

1. 八位二进制比较器的 VHDL 描述的代码示例

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
```

```
ENTITY comp8 IS
PORT (a,b   : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
      fa, fb, fe   : OUT STD_LOGIC);
END comp8;
ARCHITECTURE one OF comp8 IS
BEGIN
    PROCESS(a,b)
      BEGIN               -- 下面为顺序语句,且为行为描述
        IF a > b THEN   fa <= '1';
                        fb <= '0';
                        fe <= '0';
        ELSIF a < b THEN   fa <= '0';
                        fb <= '1';
                        fe <= '0';
        ELSIF a = b THEN   fa <= '0';
                        fb <= '0';
                        fe <= '1';
        END IF;
    END PROCESS;
END one;
```

2. 八位二进制比较器的 Verilog HDL 描述

```
module  comp8v(a, b, fa, fb, fe);
input[7:0]     a, b;
output     fa, fb, fe;
reg[7:0]     fa, fb, fe;
always( * )
    begin
        if (a > b)  begin  fa = 1;  fb = 0;  fe = 0;   end
        else if (a < b)  begin fa = 0; fb = 1; fe = 0;  end
        else if (a == b)  begin  fa = 0;  fb = 0;  fe = 1;  end
    end
endmodule
```

上面两段代码示例都是行为描述,具有一定编程基础的读者都很容易理解其功能。

6.2.4 JK 触发器设计

JK 触发器是一种时序电路,其原理图如图 6-4 所示。其中 J、K 是数据输入端,CLR 是复位控制输入端。当 CLR=0 时,触发器的状态被置为 0 态。CLK 是时钟输入端,Q 和 QN 是触发器的两个互补输出端。

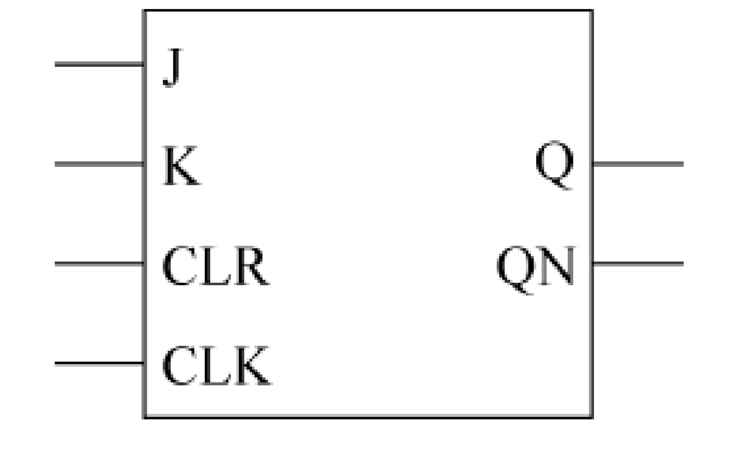


图 6-4 JK 触发器原理图

1. JK 触发器 VHDL 描述的代码示例

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY myjkff IS
```

```
PORT(j, k, clr: IN STD_LOGIC;
    clk: IN STD_LOGIC;
    q, qn: BUFFER STD_LOGIC);                    --带缓冲的输出端口
END myjkff;
ARCHITECTURE one OF myjkff IS
BEGIN
    PROCESS(j, k, clr, clk)
    VARIABLE jk: STD_LOGIC_VECTOR(1 DOWNTO 0);   --jk是变量
                                                 --只在process内部有效
    BEGIN
     jk:= (j & k);
     IF clr='0' THEN   q<='0';    qn<='1';
     ELSIF clk'EVENT AND clk='0' THEN            --clk时钟信号的下降沿
       CASE jk IS
          WHEN "00" => q  <=q;      qn<= qn;
          WHEN "01" => q  <= '0';   qn<= '1';
          WHEN "10" => q  <= '1';       qn<= '0';
          WHEN "11" => q  <= NOT q;  qn<= NOT qn;
          WHEN OTHERS =>  NULL;
       END CASE ;
     END IF;
    END PROCESS;
END one;
```

2. JK触发器 Verilog HDL 描述的代码示例

```
module myjkff(j, k, clr, clk, q, qn);
input         j, k, clr, clk;
output        q, qn;
reg           q, qn;
always @(negedge clr or negedge clk)          //下降沿触发
begin
    if (~clr)  begin q = 0;  qn = 1; end
    else
         case ({j,k})
            2'b00:  begin q = q;   qn = qn;   end
            2'b01:  begin q = 0;   qn = 1;    end
            2'b10:  begin q = 1;   qn = 0;    end
            2'b11:  begin q = ~q;   qn = ~qn;  end
            default begin q = 0;   qn = 1;    end
         endcase
end
endmodule
```

6.3　层次化设计

6.3.1　描述方式

在介绍层次化设计之前，首先介绍一下 HDL 的描述方式，就是采用 HDL 如何描述一个具体的硬件。描述方式总体上可以分为三种：行为描述、结构描述及混合描述。

1. 行为描述

行为描述是指硬件模块的输入/输出的响应情况或其行为，而不指定具体的硬件结构。图 6-5 所示是对该模块的描述，只是关注其输出对输入的处理行为，而不关注其具体的硬件细节。这种描述方式的优点是对硬件细节要求较低，方便用户设计，方便通过阅读代码理解硬件功能。但其局限性是，不能描述过于复杂的系统，如果描述过于复杂的系统会造成不能综合的结果，也就是 EDA 工具无法根据用户的行为描述自动生成对应的真实硬件。所以行为描述的结果可用于仿真，对于不太复杂的模块，也可以用于综合。

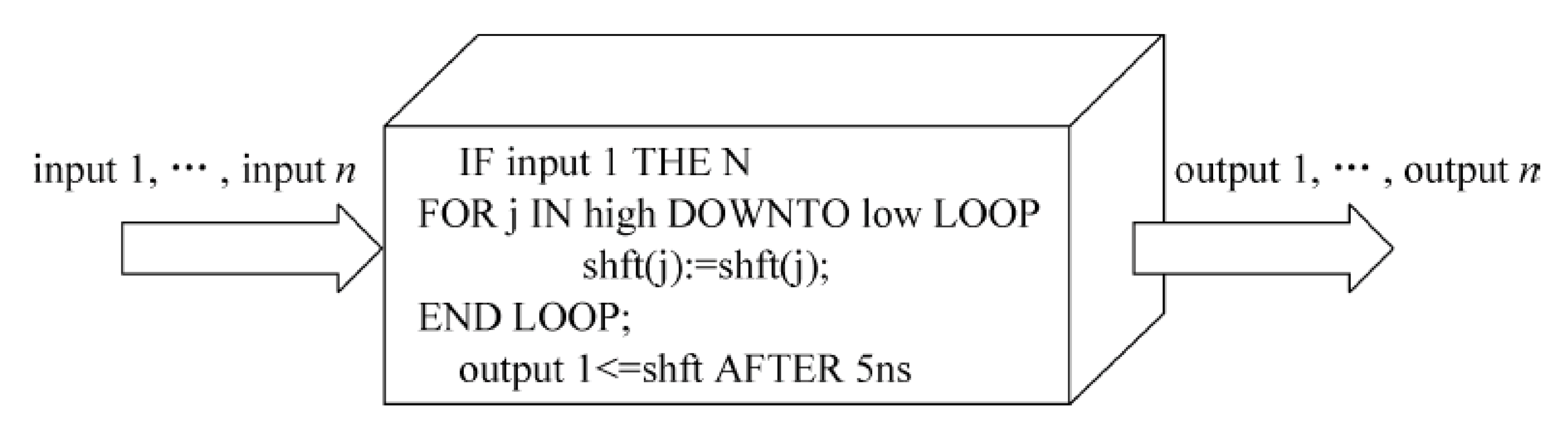


图 6-5 行为描述示例

2. 结构描述

和行为描述相对应，结构描述关注的是模块的结构，即如何用更低级的子模块通过特定的结构形成需要的模块。图 6-6 所示顶层模块的功能有两个非门以及子模块 1、子模块 2 组合完成。它的优点是由于给出了硬件的构成，EDA 工具总能对其进行综合生成具体的硬件。但缺点是，由于要考虑具体的硬件架构，对硬件基础的要求较高，且涉及太多细节，导致开发效率低。而且对于硬件基础不是特别好的用户，通过阅读代码很难理解硬件完成的功能。

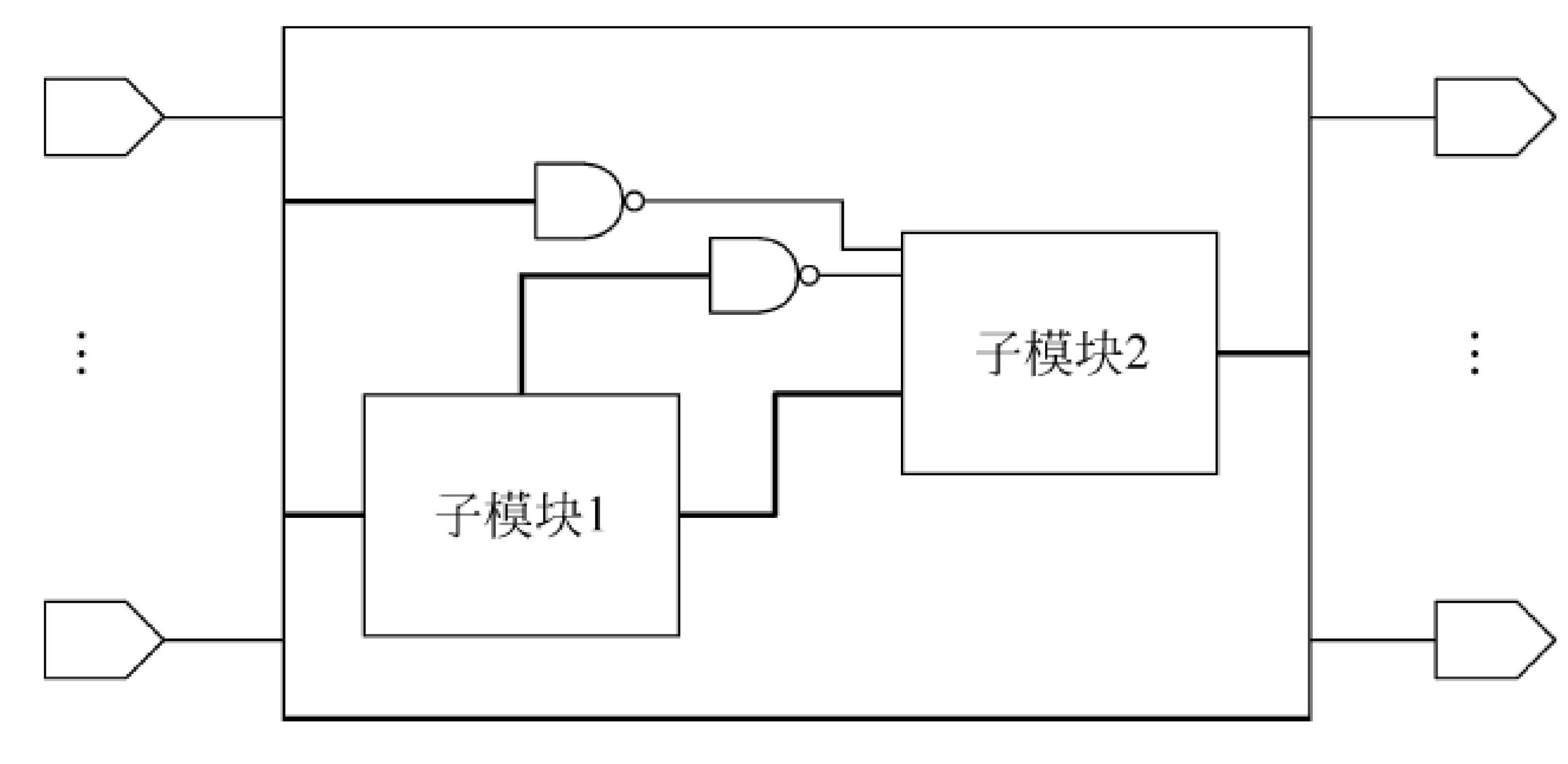


图 6-6 结构描述

3. 混合描述

混合描述是上述两种描述方式的结合使用。对于大型复杂的系统，首先在顶层采用结构描述的方式进行顶层的功能划分和子模块功能定义。逐级进行功能的细化，直到采用行为描述可以较好完成的基本功能，则采用行为描述完成。因此就会设计到在一个功能模块的内部嵌套调用更低的若干子模块，这就涉及层次化设计。

6.3.2 层次化设计的写法

层次化设计就是当一个模块的功能较为复杂，而需要把它拆成若干子模块并在上一级模块调用的设计方法。如图 6-6 所示，就是如何在顶层模块调用子模块 1 和子模块 2 的设

计，就是层次化设计。

1. 层次化设计的 VHDL 描述

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY Topmodule IS
port(       a, b: IN integer range 0 to 255;
        q: OUT integer range   0 to 65535);
END Topmodule ;
ARCHITECTURE one of Topmodule   is
  signal nea : integer range 0 to 255;          --此处先声明内部所用的信号
  signal nec : integer range 0 to 255;
  signal c    : integer range 0 to 255;
  signal d    : integer range 0 to 255;

  component Submodule1                          --声明子模块 1
        port (       in1  : in   integer range 0 to 255;
             out1 : out   integer range 0 to 255;
             out2 : out   integer range 0 to 255
        );
  end component;

  component Submodule2                          --声明子模块 2
    port (       in1  : in   integer range 0 to 255;
             in2  : in   integer range 0 to 255;
             in3  : in   integer range 0 to 255;
             out1 : out   integer range 0 to 255
        );
  end component;

BEGIN
  nea <= not a;                                 --进行信号的赋值，是并行语句
  nec <= not c;
  submod1 : Submodule1                          --使用子模块 1，将其实例化
    port map   (         in1  => a,             --并和相关信号连接
                  out1  => c,
                  out2  => d);

  Submod2 : Submodule2                          --使用子模块 2
    port map (    in1   => nea,
                  in2   => nec,
                  in3   => d,
                  out1  => q);
END one;
```

如上所述，VHDL 代码中需要用到子模块时，需要在 Architecture…Begin 之间声明子模块，说明其输入/输出端口。

具体调用子模块，需要在 Begin…End 之间将声明的子模块实例化，可以根据需要实例化多个。并将其输入/输出端口和相关的信号关联起来，此时子模块就不是一个孤立的模

块,而是可以和其他系统协同工作了。

2. 层次化设计的 Verilog HDL 描述

```
module Topmodule (a, b, q);
  input [7:0]   a, b;
  output [15:0]   q;
  wire [7: 0]  nea, nec, c, d;
    assign nea = not a;              //进行信号的赋值,是并行语句
    assign nec = not c;
     submod1  Submodule1             //使用子模块 1,将其实例化
        (   .in1(a),                 //并和相关信号连接
            .out1(c),
            .out2(d));
     submod2  Submodule2
        (   .in1(nea),
            .in2(nec),
            .in3(d),
            .out1(q));
Endmodule
```

如上所述,Verilog HDL 代码中需要用到子模块时,无须声明直接调用即可。具体调用子模块,对其进行例化并将其输入/输出端口和相关的信号关联起来即可。

通过上面的介绍,在完成实验设计时,就可以通过把一个复杂的系统划分成子模块,最终通过层次化的方式完成整个系统。

上面介绍了 HDL 的功能模块的基本结构和设计方法,以及利用层次化设计方法设计更复杂的系统,下面简单介绍一下 VHDL 和 Verilog HDL 的语言基础。

6.4 VHDL 语言基础

6.4.1 标识符

标识符用来定义常数、变量、信号、端口、子程序或参数的名字,由字母(A～Z,a～z)、数字(0～9)和下画线(_)字符组成。VHDL 规定首字符必须是字母,末字符不能为下画线。不允许出现两个连续的下划线。不区分大小写。VHDL 定义的保留字(关键字)不能用作标识符,标识符字符最长可以是 32 个字符。

6.4.2 数据对象

常用的数据对象有常量、变量和信号:

(1) 常量是对某一常量名赋予一个固定的值,而且只能赋值一次。通常赋值在程序开始前进行,该值的数据类型则在说明语句中指明。用法如下:

```
Constant  常数名: 数据类型: = 表达式
Constant  Vcc: real: = 5.0;               -- 定义 Vcc 的数据类型是实数,赋值为 5.0V
```

(2) 变量只能在进程语句、函数语句和过程语句结构中使用。变量的赋值是直接的,非

预设的。分配给变量的值立即成为当前值。变量不能表达"连线"或存储元件,不能设置传输延时量。

Variable　变量名:数据类型　:=初始值;

```
Variable count: integer 0 to 255:=20; --定义 count 整数变量,变化范围 0~255,初始值为 20
```

变量赋值语句:目标变量名　:=　表达式;

```
x:=10.0;                                --实数变量赋值为 10.0
```

(3) 信号表示逻辑门的输入或输出,类似于连接线。也可以表达存储元件的状态。信号通常在构造体、程序包和实体中说明。

Signal　信号名:数据类型 :=初始值

```
Signal clock: bit :='0';                --定义时钟信号类型,初始值为 0
```

信号赋值语句:目标信号名　<=　表达式;

```
x<=9;
Z<=x after 5 ns;                        --在 5ns 后将 x 的值赋予 z
```

6.4.3　数据类型

数据类型包括以下 9 种。

1. 布尔:(Boolean)

```
TYPE BOOLEAN IS (FALSE, TRUE);   --取值为 FALSE 和 TRUE,不是数值,不能运算,一般用于关系运算符
```

2. 位:(Bit)

```
TYPE BIT IS ('0','1');                           --取值为 0 和 1,用于逻辑运算
```

3. 位矢量:(Bit_Vector)

```
TYPE BIT_VECTOR IS ARRAY (Natural range<>) OF BIT;  --基于 Bit 类型的数组,用于逻辑运算
SIGNAL a: Bit_Vector(0 TO 7);                       --第 0 位在最左边
SIGNAL a: Bit_Vector ( 7 DOWNTO 0);                 --第 0 位在最右边
```

4. 字符:(Character)

```
TYPE CHARACTER IS (NUL, SOH,STX, …, ' ', '!', …);   --通常用单引号''引起来,区分大小写
```

5. 字符串:(String)

```
VARIABLE string_var: STRING (1 TO 7);
string_var:="A B C D" ;                             --通常用双引号""引起来,区分大小写
```

6. 整数:(Integer)

取值范围 $-(2^{31}-1)\sim(2^{31}-1)$,可用 32 位有符号的二进制数表示,如:

```
variable a: integer range -63 to 63
```

7. 实数：（Real）

取值范围为－1.0E38～＋1.0E38，仅用于仿真不可综合。

```
1.0                    -- 十进制浮点数
8#43.6#e+4             -- 八进制浮点数
43.6E-4                -- 十进制浮点数
```

8. 时间：（Time）

物理量数据，完整的包括整数和单位两个部分，用至少一个空格隔开，仅用于仿真不可综合。分别有飞秒 fs、皮秒 ps、纳秒 ns、微秒 μs、毫秒 ms、秒 sec、分 min、时 hr。

9. 错误等级：（Severity Level）

表示系统状态，仅用于仿真不可综合；

```
TYPE severity_level IS (NOTE、WARNING、ERROR、FAILURE);
```

6.4.4 数据类型转换

数据类型转换包括以下几种：

1. 类型标记法

```
Variable A: integer;     Variable B: real;
A = integer (B);     B = real (A);
```

2. 函数法

```
Conv_interger (A);          -- 由 std_logic 转换为 integer 型,在 std_logic_unsigned 包
```

3. 常数转换法/常量转换法

```
Type conv_table is array(std_logic) of bit;        -- 定义 cony_table 的数据类型为数组
Constant table:conv_table: = ('0'|'L' =>'0','1'|'H' =>'1',others =>'0');
                                                   -- Constant table 为具有转换表性质的常数
Signal a: bit; signal b: std_logic;                -- 定义 a,b 类型
A <= table(b);                                     -- 将 std_logic 型转换为 bit 型
```

4. 属性

属性提供的是关于信号、类型等的指定特性，一般常用有如下三种属性：

（1）'event：若属性对象有事件发生，则生成布尔值 true，常用来检查时钟边沿是否有效。检查时钟上升沿：Clock' EVENT AND Clock＝'1'。

（2）'range：用于生成一个限制性数组对象的范围。设置数据范围：'range,"0 to n"；'reverse_range,"n downto 0"。

（3）'left：生成数据类型或数据子类型的左边界值；'right：生成数据类型或数据子类

型的右边界值；'high：生成数据类型或数据子类型的上限值；'low：生成数据类型或数据子类型的下限值；'length：生成数据类型或数据子类型的长度值。

6.4.5 运算符

运算符包括算术运算符等 7 种，详细如下：

1. 算术运算符

(1) 加(+)，减(-)，乘(*)，除(/)。

(2) 取模(MOD)，取余(REM)：

取余运算(a REM b)的符号与 a 相同，其绝对值小于 b 的绝对值。

例如：(-5)REM 2=(-1)　　5 REM 2=(1)

取模运算(a MOD b)的符号与 b 相同，其绝对值小于 b 的绝对值。

例如：(-5)MOD 2=1　　5 MOD (- 2)=(-1)

(3) SLL：将位向量左移，右边移空位补零。

(4) SRL：将位向量右移，左边移空位补零。

(5) SLA：将位向量左移，右边第一位的数值保持原值不变。

(6) SRA：将位向量右移，左边第一位的数值保持原值不变。

(7) ROL 和 ROR：自循环左右移位。

例：

```
"1100"SLL1 = "1000"  -- 左移空位补零        "1100"SRL1 = "0110"  -- 右移空位补零
"1100"SLA1 = "1000"  -- 左移最右值不变      "1100"SRA1 = "1110"  -- 右移最左值不变
"1100"ROL1 = "1001"  -- 循环左移            "1100"ROR1 = "0110"  -- 循环右移
```

(8) 乘方(**)，绝对值(ABS)。

2. 关系运算符

等于(=)，不等于(/=)，小于(<)，大于(>)，小于等于(<=)，大于等于(>=)。

3. 逻辑运算符

(1) 与(AND)，或(OR)。

(2) 与非(NAND)，或非(NOR)。

(3) 异或非(XNOR)，非(NOT)，异或(XOR)。

4. 赋值运算符

(1) 为信号赋值<=。

(2) 为变量赋值:=。

5. 关联运算符

=>关联运算符用来将子模块的输入/输出端口与上层模块中的信号关联起来。

6. 并置操作符

& 并置操作符用来将信号合并成更大宽度的信号。

```
SIGNAL a : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
SIGNAL d : STD_LOGIC_VECTOR (1 DOWNTO 0) ;
```

```
a <= '1'&'0'&d(1)&'1' ;          -- 元素与元素并置,并置后的数组长度为 4
IF a & d = "101011" THEN…        -- 在 IF 条件句中可以使用并置符
```

7. 符号运算符

正(＋),负(－)。

6.4.6 运算符优先级

VHDL 的运算符优先级从高到低分别是(同一行中左高右低):

逻辑运算符:NOT。

算术运算符:ABS, ** ,REM,MOD,/, * ,－(负号),＋(正号)。

并置运算符:&。

算术运算符:－(减法),＋(加法)。

逻辑运算符:ROR,ROL,SRA,SLA,SRL,SLL。

关系运算符:＞＝,＞,＜＝,＜,/＝,＝。

逻辑运算符:XNOR,XOR,NOR,NAND,OR,AND。

6.4.7 VHDL 常用语法

VHDL 常用语法如下:

1. 简单赋值语句

目标信号名＜＝表达式,举例如下:

```
ARCHITECTURE Behavior OF FreDevider IS
SIGNAL Clk:Std_Logic;
BEGIN
    PROCESS(Clock)
BEGIN
    IF rising_edge(Clock) THEN
         Clk <= NOT Clk;
    END IF;
END PROCESS;
Clkout <= Clk;
```

2. 选择信号赋值语句

```
WITH 选择表达式 SELECT
    赋值目标信号 <=          表达式 1    WHEN   选择值 1,
                            表达式 2    WHEN   选择值 1,
                            表达式 n    WHEN   OTHERS;
```

选择值要覆盖所有可能情况,若不能一一指定,用 OTHERS 为其他情况找个出口;选择值必须互斥,不能出现条件重复或重叠的情况。

举例如下:

```
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL;
ENTITY MUX IS
PORT
```

```
( Data0,Data1,Data2,Data3: IN Std_Logic_VECTOR(7 DOWNTO 0);
  Sel: IN Std_Logic_Vector(1 DOWNTO 0);
  DOUT: OUT Std_Logic_Vector(7 DOWNTO 0)
);
END;
ARCHITECTURE DataFlow OF MUX IS
BEGIN
    WITH Sel SELECT
        DOUT <=   Data0  WHEN "00",
                         Data1  WHEN "01",
                         Data2  WHEN "10",
                         Data3  WHEN "11",
"00000000" WHEN OTHERS;
END;
```

3. 选择信号赋值语句

```
赋值目标信号 <=  表达式 1   WHEN  赋值条件 1  ELSE
                 表达式 2   WHEN  赋值条件 2  ELSE
                 表达式 n   WHEN  赋值条件 n  ELSE
                 表达式;
```

各赋值语句有优先级的差别,按书写顺序从高到低排列;各赋值条件可以重叠。举例如下:

```
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL;
ENTITY Priority_Encoder IS
PORT
( I: IN Std_Logic_VECTOR(7 DOWNTO 0);
  A: OUT Std_Logic_Vector(2 DOWNTO 0)
);
END;
ARCHITECTURE DataFlow OF Priority_Encoder IS
BEGIN
        A <= "111"  WHEN I(7) = '1'  ELSE
               "110"  WHEN I(6) = '1'  ELSE
               "101"  WHEN I(5) = '1'  ELSE
               "100"  WHEN I(4) = '1'  ELSE
               "011"  WHEN I(3) = '1'  ELSE
               "010"  WHEN I(2) = '1'  ELSE
               "001"  WHEN I(1) = '1'  ELSE
               "000"  WHEN I(0) = '1'  ELSE
               "111";
END;
```

4. 进程(PROCESS)语句

进程语句定义顺序语句模块,用于将从外部获得的信号值,或内部的运算数据向其他的

信号进行赋值。多个进程之间是并行的，但进程内部是顺序语句。进程只有在特定的时刻（敏感信号发生变化时）才会被激活。

```
[进程标号: ] PROCESS (敏感信号参数表)
[声明区];
   BEGIN
      顺序语句
END PROCESS  [进程标号];
```

举例如下：

```
ARCHITECTURE Behavior OF FreDevider IS
SIGNAL Clk:Std_Logic;
BEGIN
    PROCESS(Clock)
BEGIN
    IF rising_edge(Clock) THEN
         Clk <= NOT Clk;
    END IF;
END PROCESS;
Clkout <= Clk;
```

5. 顺序语句

顺序语句仅出现在进程和子程序中。顺序语句综合后，映射成实际的门电路，系统一上电，门电路开始工作。电路可实现逻辑上的顺序执行，实际上所有门电路都是并行工作的。

（1）赋值语句，举例如下：

```
ENTITY TEST_Signal IS
PORT
( Reset, Clock: IN Std_logic;
  NumA, NumB: OUT Integer RANGE 0 TO 255
);
END;
ARCHITECTURE TEST OF TEST_Signal IS
      SIGNAL A, B: Integer RANGE 0 TO 255;
  BEGIN
      PROCESS (RESET,Clock)
      VARIABLE C: Integer RANGE 0 TO 255;
BEGIN
         IF RESET = '1' THEN
              A <= 0; B <= 2;C: = 0;
         ELSEIF rising_edge(Clock) THEN
             C: = C + 1;A <= C + 1;B <= A + 2;
         END IF;
      END PROCESS;
      Num A <= A; NumB <= B;
END;
```

(2) IF 语句,举例如下:

```
IF 条件式 THEN
    顺序语句
END IF;
```

```
IF 条件式 THEN
    顺序语句
ELSE
    顺序语句
END IF;
```

```
IF 条件式 THEN
    顺序语句
ELSEIF 条件式 2
THEN
    顺序语句
ELSE
    顺序语句
END IF;
```

```
ENTITY COMP_GOOD IS
 PORT(a1   :   IN BIT;
        b1   :   IN BIT;
        q1   : OUT BIT   );
END ;
ARCHITECTURE one OF COMP_GOOD IS
  BEGIN
   PROCESS (a1,b1)
    BEGIN
     IF   a1 > b1   THEN   q1 <= '1';
     ELSE    q1 <=  '0';
     END IF;
   END PROCESS ;
 END ;
```

(3) Case 语句,举例如下:

```
CASE 表达式 IS
  WHEN 选择值[|选择值 ]=>顺序语句;
  WHEN 选择值[|选择值 ]=>顺序语句;
  WHEN OTHERS =>顺序语句;
END CASE;
```

选择值不可重复或重叠;当 CASE 语句的选择值无法覆盖所有的情况时,要用 OTHERS 指定未能列出的其他所有情况的输出值。

```
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL;
ENTITY MUX IS
PORT
( Data0,Data1,Data2,Data3: IN Std_Logic_VECTOR(7 DOWNTO 0);
  Sel: IN Std_Logic_Vector(1 DOWNTO 0);
  DOUT: OUT Std_Logic_Vector(7 DOWNTO 0)
);
END;
 ARCHITECTURE DataFlow OF MUX IS
    BEGIN
```

```
        CASE Sel IS
            WHEN "00" => DOUT <=  Data0;
            WHEN "01" => DOUT <=  Data1;
            WHEN "10" => DOUT <=  Data2;
            WHEN "11" => DOUT <=  Data3;
            WHEN OTHERS  => DOUT <= "00000000";
        END CASE;
    END PROCESS;
END;
```

(4) LOOP 语句,举例如下:

```
[LOOP 标号:] FOR 循环变量 IN 循环次数范围 LOOP
      顺序语句
   END LOOP [LOOP 标号];
```

```
Sum: = 0;
FOR i  IN 0 TO 9 LOOP
   Sum: = Sum + i;
END LOOP;
```

(5) NEXT 语句,举例如下:

NEXT 语句主要用在 LOOP 语句执行中有条件或无条件转向控制,跳向 LOOP 语句的起点。

NEXT [循环标号][WHEN 条件];

- NEXT;——无条件终止当前循环,跳回到本次循环 LOOP 语句处,开始下一次循环。
- NEXT LOOP 标号;——当有多重 LOOP 语句嵌套时,跳转到指定标号 LOOP 语句处,重新开始执行循环操作。
- NEXT LOOP 标号 WHEN 条件表达式;——条件表达式为 TRUE,执行 NEXT 语句,进入跳转操作,否则向下执行。

(6) EXIT 语句:EXIT 语句主要用在 LOOP 语句执行中有条件或无条件内部转向控制,跳向 LOOP 语句的终点,用于退出循环。当程序需要处理保护、出错和警告状态时,EXIT 语句能提供一个快捷、简便的方法。

EXIT [循环标号][WHEN 条件];

- EXIT;——无条件从当前循环中退出。
- EXIT LOOP 标号;——程序执行退出动作无条件从循环标号所标明的循环中退出。
- EXIT LOOP 标号 WHEN 条件表达式;——条件表达式为 TRUE,程序从当前循环中退出。

(7) NULL 语句。

NULL 为空操作语句,一般用于 CASE 中,表示在某些情况下对输出不作任何改变,隐含锁存信号。不能用于纯组合逻辑电路。举例说明如下:

```
PROCESS (Clock)
BEGIN
```

```
    IF rising_edge (Clock)   THEN

      CASE Sel IS
        WHEN…
        WHEN OTHERS => NULL;
      END CASE;
    END IF;
END PROCESS;
```

6.5　Verilog HDL 语言基础

6.5.1　数据类型

wire 型数据常用来表示用于以 assign 关键字赋值的组合逻辑信号。Verilog 的 module 中输入/输出信号类型默认时自动定义为 wire 型。wire 型信号可以用作任何表达式的输入，也可以用作 assign 语句或实例元件的输出。用法与 VHDL 中的 signal 对应。

寄存器是数据储存单元的抽象，寄存器数据类型的关键字是 reg。通过赋值语句可以改变寄存器储存的值，其作用与改变触发器储存的值相当。reg 型数据常用来表示用于 always 模块内的指定信号，常代表触发器。通常，在设计中要由 always 块通过使用行为描述语句来表达逻辑关系。在 always 块内被赋值的每一个信号都必须定义成 reg 型，reg 类型数据的默认初始值为不定值'X'。用法与 VHDL 中的 variable 对应。

```
reg  [width-1 : 0] R变量1,R变量2;
wire [width-1 : 0] W变量1,W变量2;
```

例：

```
reg  [31:0]   out;                      //定义了一个32位的名为out的reg型数据
reg           rega;                     //定义了一个一位的名为rega的reg型数据
wire          ifu2biu_icb_cmd_valid;    //定义了一个一位的wire数据
wire          ifu2biu_icb_cmd_ready;
wire [31:0]   ifu2biu_icb_cmd_addr;     //定义了一个32位的wire数据
```

6.5.2　数字表示形式

数字表达方式有以下三种：

(1) <位宽><进制><数字>，这是比较全面的描述方式。

(2) <进制><数字>，在这种描述方式中，数字的位宽采用默认位宽。

(3) <数字>，在这种描述方式中，采用默认进制十进制。

对于<进制>的表示：二进制整数用 b 或 B 表示，十进制整数用 d 或 D 表示，十六进制整数用 h 或 H 表示，八进制整数用 o 或 O 表示。

具体用法示例：

```
8'b10101111     //位宽为8的数的二进制表示, 'b表示二进制
4'ha            //位宽为4的数的十六进制,'h表示十六进制
```

下画线(underscore)可以用来分隔开数的表达以提高程序可读性。但不可以用在位宽和进制处,只能用在具体的数字之间。见下例:

```
16'b1010_1011_1111_1010        //合法格式
8'b_0011_1010                  //非法格式
```

6.5.3 parameter 定义常量

parameter 可用来定义常量,即用 parameter 来定义一个标识符代表一个常量,称为符号常量即标识符形式的常量。采用标识符代表一个常量可提高程序的可读性和可维护性。

parameter 型数据是一种常数型的数据,其说明格式如下:

```
parameter 参数名 1 = 表达式,参数名 2 = 表达式, …;
```

具体用法示例如下:

```
parameter DW = 32;               //定义参数 DW 为常量 32
parameter e = 25, f = 29;        //定义二个常数参数
parameter r = 5.7;               //声明 r 为一个实型参数
```

6.5.4 宏定义'define

用一个指定的标识符(即名字)来代表一个字符串,它的一般形式为:

```
'define 标识符(宏名) 字符串(宏内容)
```

其作用是用指定的宏名来代替后面的宏内容,宏名一般使用大写字母以区别变量名。例:

```
'define E203_ADDR_SIZE     16   //用 E203_ADDR_SIZE 来代替 16
'define E203_PC_SIZE       16   //用 E203_PC_SIZE 来代替 16
```

在引用已定义的宏名时,必须在宏名的前面加上符号“'”,表示该名字是一个经过宏定义的名字。例:

```
output ['E203_PC_SIZE - 1:0] inspect_pc
```

6.5.5 运算符及表达式

Verilog HDL 语言的运算符范围很广,其运算符按其功能可分为以下几类。

1. 算术运算符(+, -, *, /, %)

(1) + (加法运算符,或正值运算符,如 rega+regb,+3);

(2) - (减法运算符,或负值运算符,如 rega-3,-3);

(3) * (乘法运算符,如 rega * 3);

(4) / (除法运算符,如 5/3);

(5) %(模运算符,或称为求余运算符,要求%两侧均为整型数据。如 7%3 的值为 1)。

2. 赋值运算符(=,<=)

信号有两种赋值方式:

(1) 非阻塞(Non_Blocking)赋值方式(<=):

① 块结束后才完成赋值操作。

② b 的值并不是立刻就改变的。

③ 这是一种比较常用的赋值方法(绝大多数情况下使用非阻塞赋值)。

(2) 阻塞(Blocking)赋值方式(=):

① 赋值语句执行完后,块才结束。

② b 的值在赋值语句执行完后立刻就改变的。

③ 可能会产生意想不到的结果。

如图 6-7 给出示例说明:

```
首先假设 a 等于 0
    always@(*)                    always@(*)
    begin                         begin
        ...                           ...
        a <= a + 1;                   a = a + 1;
        b <= a + 1;                   b = a + 1;
    end                           end
    执行后 a 为 1,b 为 1          执行后 a 为 1,b 为 2
```

图 6-7 阻塞和非阻塞赋值比较

3. 关系运算符(>,<,>=,<=)

a < b:a 小于 b

a > b:a 大于 b

a <= b:a 小于或等于 b

a >= b:a 大于或等于 b

4. 逻辑运算符(&&,||,!)

(1) && 逻辑与

(2) || 逻辑或

(3) ! 逻辑非

"&&"和"||"是二目运算符,它要求有两个操作数,如:(a>b)&&(b>c)或(a<b)||(b<c)。

"!"是单目运算符,只要求一个操作数,如:!(a>b)。逻辑 & 和逻辑 | 是双目运算符,逻辑非是单目运算符。如果操作数是多位的,则将操作数看作整体,若操作数中每一位都是 0 值则为逻辑 0 值,若操作数当中有 1,则为逻辑 1 值。

5. 条件运算符(?:)

```
assign A = B ? C : D          //如果 B 为 1 则将 C 赋给 A,如果 B 不为 1 将 D 赋给 A
```

6. 位运算符(~，|，^，&，^~)

(1)"取反"运算符~：~是一个单目运算符，用来对一个操作数进行按位取反运算。

(2)"按位与"运算符 &：按位与运算就是将两个操作数的相应位进行与运算。

(3)"按位或"运算符|：按位或运算就是将两个操作数的相应位进行或运算。

(4)"按位异或"运算符^(也称之为 XOR 运算符)：按位异或运算就是将两个操作数的相应位进行异或运算。

(5)"按位同或"运算符^~：按位同或运算就是将两个操作数的相应位先进行异或运算再进行非运算。

(6)不同长度的数据进行位运算：两个长度不同的数据进行位运算时，系统会自动将两者按右端对齐。位数少的操作数会在相应的高位用 0 填满，以使两个操作数按位进行操作。

7. 移位运算符(<<,>>)

<< (左移位运算符)和>>(右移位运算符)。其使用方法如下：

```
a >> n 或 a << n
```

a 代表要进行移位的操作数，n 代表要移几位。这两种移位运算都用 0 来填补移出的空位。例：

```
4'b1001 << 2  =  6'b100100;
4'b1011 << 2  =  6'b101100;
4'b1101 >> 1  =  4'b0110;
```

8. 拼接运算符({ })

这个运算符可以把两个或多个信号的某些位拼接起来进行运算操作。例：

```
qout_r <= {16{1'b1}};                //对 qout_r 的 16 位每位赋 1
```

6.5.6 运算符优先级

如图 6-8 是 Verilog HDL 中的运算符优先级。

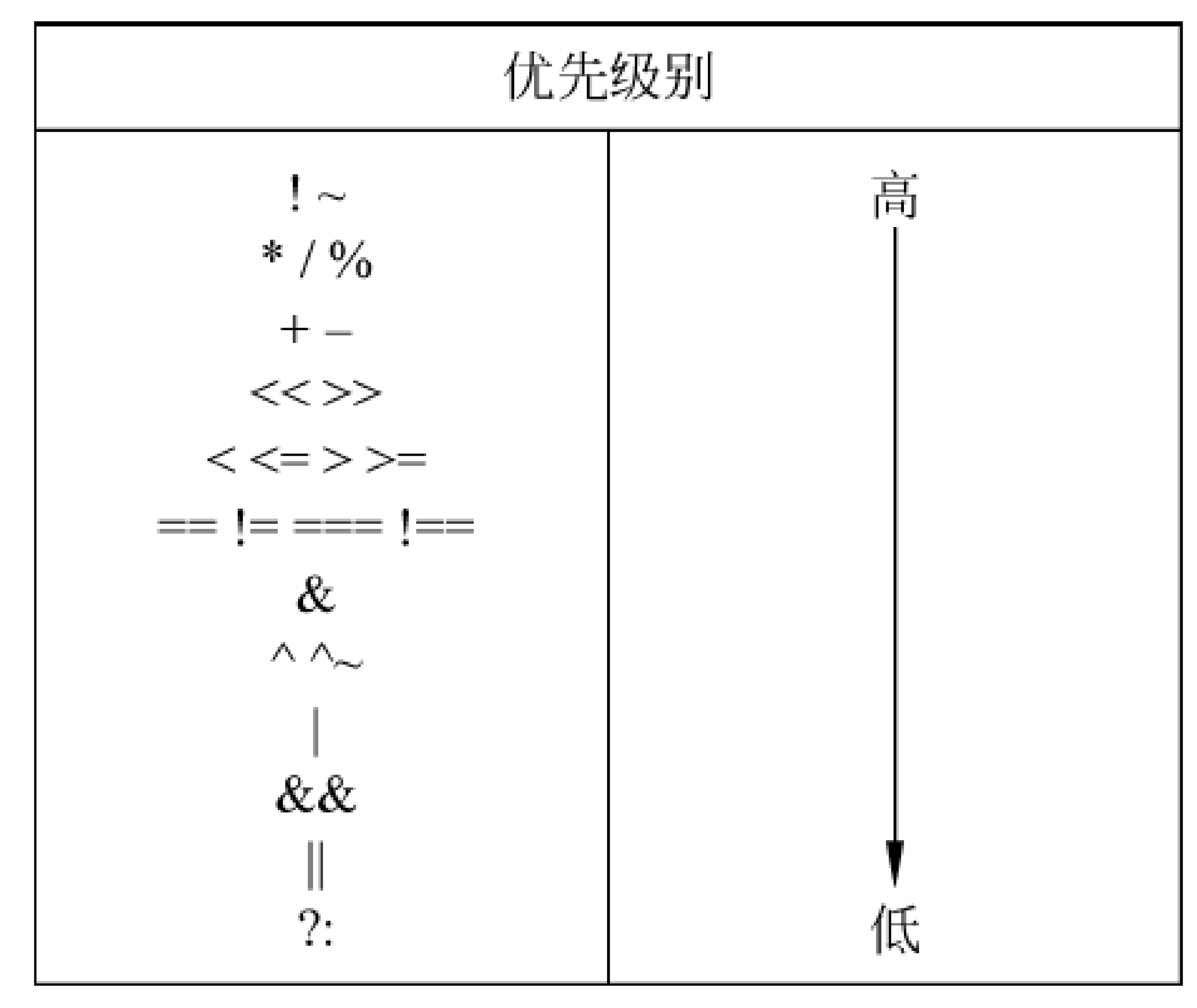


图 6-8 运算符优先级

6.5.7 Verilog HDL 常用语法

1. if_else 语句

if 语句是用来判断所给定的条件是否满足，根据判定的结果（真或假）决定执行给出两种操作中的哪一个。常用格式为三种：

(1) if(表达式)　　语句序列 1；

(2) if(表达式)　　语句序列 1；
　　else　　语句序列 2；

(3) if(表达式 1)　　语句序列 1；
　　else if(表达式 2)　　语句序列 2；
　　…
　　else if(表达式 n)　　语句序列 n；
　　else　　语句序列 n+1；

举例说明如下：

```
1) always@(*) begin
       if (!clk_in)
           en = (clock_en | test_mode);
   end
   assign clk_out = enb & clk_in;
2) if(DEEP == 0)             o_vld = i_vld1;
   else if(DEEP == 1)        o_vld = i_vld2;
   else                      o_vld = 0;
```

2. case 语句

case 语句是一种多分支选择语句，if 语句只有两个分支可供选择，当实际问题中需要多分支选择时，可使用 case 语句直接处理多分支选择。

```
case(表达式)
    值 1:      语句序列 1;
    值 2:      语句序列 2;
    …
    值 n:      语句序列 n;
    default:  语句序列 n+1;
endcase
```

举例说明如下：

```
case(type)
    1'b1:    out <= a + b;
    1'b0:    out <= a - b;
    default: out <= 0;
endcase
```

3. for 语句

for 语句是循环语句，一般形式为：

for(表达式 1; 表达式 2; 表达式 3)

其最简单的应用如下所示：

for(循环变量赋初值; 循环结束条件; 循环变量增值)

举例说明如下：

```
for (i = 0 ; i < 10 ; i = i + 1) begin
assign fifo_rf_en[i]  =  wen & wptr_vec_r[i];
    sirv_gnrl_dffl  fifo_rf_dffl(fifo_rf_en[i], i_dat, fifo_rf_r[i], clk);
end
```

第 7 章 CHAPTER 7 基于开源 CPU 的组成原理实验

随着教指委"计算机类系统能力培养"工作的推进，越来越多高校的计算机组成原理实验环节开始基于 FPGA 采用 Verilog/VHDL 进行，师资力量比较强的高校开始让学生设计兼容 MIPS 或 ARM 指令集的处理器，甚至要求自己的处理器上跑自己的操作系统。这些做法极大地提升了学生对计算机软硬件整体系统的理解。

然而，我们也应该看到，因为各种条件的限制，目前各高校要求学生基于 FPGA 用 HDL 设计处理器时，关注点主要在于学生对计算机组成原理理论的掌握情况及逻辑功能的设计和实现，考查的主要是学生在理论指导下从无到有或增量式的系统设计能力，比如一条指令的数据通路设计。进而学会逐步地增加指令及其对应的数据通路，完成整个处理器的设计。这种方式对于学生掌握处理器的设计原理对于工业界的设计经验和设计规范借鉴的不太多。

目前随着开源指令集 RISC-V 的快速发展，涌现了若干工业界人士推出的开源 RISC-V 处理器。个人认为这些体现工程考虑的开源项目如能以合适的方式引入高校组成原理实验中，将会对我们老师及学生的实际工程能力有极大的帮助。同时，开源开放是信息技术发展的趋势，我们培养的学生也应该尽早地学习如何合理地利用开源项目完成开发需求，并基于开源项目做更多的创新工作，而不是每次都从零开始重复地造轮子。

因此，本书拟通过引入开源的 RISC-V 处理器项目，一方面让学生有机会学习来自工业界工程师们的代码设计，学习其设计理念和思想；另一方面可让学生基于开源项目做进一步的工作，甚至反过来为开源社区做贡献。当然，为了照顾到不同的需求，本书还是简单介绍了基于 RISC-V 指令集的处理器的逐条实现方法。

7.1 RISC-V 指令集

RISC-V 指令集使用模块化的方式进行组织，每个模块使用一个英文字母来表示。RISC-V 最基本也是唯一强制要求实现的指令集部分是由 I 字母表示的基本整数指令集。使用该整数指令子集，便能够实现完整的软件编译器。其他的指令子集部分均为可选的模块，具有代表性的模块包括 M/A/F/D/C，如表 7-1 所示。

表 7-1 基本指令子集

指令集	指令数	描　述
基本指令集		
RV32I	47	32 位地址空间与整数指令，支持 32 个通用整数寄存器
RV32E	47	RV32I 的子集，仅支持 16 个通用整数寄存器
RV64I	59	64 位地址空间与整数指令及一部分 32 位的整数指令
RV1128I	71	128 位地址空间与整数指令及一部分 64 位和 32 位的指令
扩展指令集		
M	8	整数乘法与除法指令
A	11	存储器原子操作指令和 Load-Reserved/Store-Conditional 指令
F	26	单精度(32 位)浮点指令
D	26	双精度(64 位)浮点指令，必须支持 F 扩展指令
C	46	压缩指令，指令长度为 16 位

下面主要针对 RV32I 指令进行介绍。想了解其他基本指令子集可以查看 RISC-V 手册。RV32I 指令子集显示了 6 种基本指令格式，分别是：用于寄存器—寄存器操作的 R 类型指令，用于短立即数和访存 load 操作的 I 型指令，用于访存 store 操作的 S 型指令，用于条件跳转操作的 B 类型指令，用于长立即数的 U 型指令和用于无条件跳转的 J 型指令。图 7-1 介绍了指令格式，图 7-2 列出了 RV32I 指令的操作码。

31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

图 7-1 RV32I 指令格式

RV32I 指令只有 6 种格式，并且所有的指令都是 32 位长，这简化了指令解码。

另外 RISC-V 指令提供 3 个寄存器操作数，而不是像 x86-32 一样，让源操作数和目的操作数共享一个字段。当一个操作本来需要有 3 个不同的操作数，但是 ISA 只提供了 2 个操作数时，编译器或者汇编程序程序员就需要多使用一条 move(搬运)指令来保存目的寄存器的值。

在 RISC-V 中，对于所有指令要读写的寄存器的标识符总是在同一位置，意味着在解码指令之前，就可以先开始访问寄存器。在许多其他的 ISA 中，某些指令字段在部分指令中被重用作为源目的地，在其他指令中又被作为目的操作数(例如，ARM-32 和 MIPS-32)，因此为了取出正确的指令字段就需要在时序本就可能紧张的解码路径上添加额外的解码逻辑，

31		25 24	20 19	15 14	12 11	7 6　　0	
imm[31:12]					rd	0110111	U lui
imm[31:12]					rd	0010111	U auipc
imm[20|10:1|11|19:12]					rd	1101111	J jal
imm[11:0]			rs1	000	rd	1100111	I jalr
imm[12|10:5]		rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]		rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]		rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]		rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]		rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]		rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]			rs1	000	rd	0000011	I lb
imm[11:0]			rs1	001	rd	0000011	I lh
imm[11:0]			rs1	010	rd	0000011	I lw
imm[11:0]			rs1	100	rd	0000011	I lbu
imm[11:0]			rs1	101	rd	0000011	I lhu
imm[11:5]		rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]		rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]		rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]			rs1	000	rd	0010011	I addi
imm[11:0]			rs1	010	rd	0010011	I slti
imm[11:0]			rs1	011	rd	0010011	I sltiu
imm[11:0]			rs1	100	rd	0010011	I xori
imm[11:0]			rs1	110	rd	0010011	I ori
imm[11:0]			rs1	111	rd	0010011	I andi
0000000		shamt	rs1	001	rd	0010011	I slli
0000000		shamt	rs1	101	rd	0010011	I srli
0100000		shamt	rs1	101	rd	0010011	I srai
0000000		rs2	rs1	000	rd	0110011	R add
0100000		rs2	rs1	000	rd	0110011	R sub
0000000		rs2	rs1	001	rd	0110011	R sll
0000000		rs2	rs1	010	rd	0110011	R slt
0000000		rs2	rs1	011	rd	0110011	R sltu
0000000		rs2	rs1	100	rd	0110011	R xor
0000000		rs2	rs1	101	rd	0110011	R srl
0100000		rs2	rs1	101	rd	0110011	R sra
0000000		rs2	rs1	110	rd	0110011	R or
0000000		rs2	rs1	111	rd	0110011	R and
0000	pred	succ	00000	000	00000	0001111	I fence
0000	0000	0000	00000	001	00000	0001111	I fence.i
000000000000			00000	00	00000	1110011	I ecall
000000000000			00000	000	00000	1110011	I ebreak
csr			rs1	001	rd	1110011	I csrrw
csr			rs1	010	rd	1110011	I csrrs
csr			rs1	011	rd	1110011	I csrrc
csr			zimm	101	rd	1110011	I csrrwi
csr			zimm	110	rd	1110011	I cssrrsi
csr			zimm	111	rd	1110011	I csrrci

图 7-2　RV32I 指令

使得解码路径的时序更为紧张。

在 RISC-V 中,立即数字段总是存在符号扩展,而且符号位总是在指令中的最高位。这意味着对于可能成为关键路径的立即数符号扩展,可以在指令解码之前进行。

7.2 基于 RISC-V 的逐条增加指令式实验

读者基于 RISC-V 指令集可以自行设计和实现处理器。下面介绍如何通过逐条增加指令,最终实现一个由 10 条指令构建的五级流水线 RISC-V 处理器。这里只涉及 RISC-V 指令集中部分的 32 位指令,通过这些基本指令使读者理解处理器流水线结构。本实验的全部代码包含在配套源代码的对应目录下。用户可以基于该处理器进一步扩展指令,完善处理器的功能。该示例处理器的设计思想参考了 OpenMIPS 处理器设计,感兴趣的读者可以参考《自己动手写 CPU》一书来继续完善此款 RISC-V 处理器。

部分立即数指令的指令说明如表 7-2 所示。

表 7-2　立即数指令说明

指令	说　明					
ADDI	imm[11:0]		rs1	000	rd	0010011
SLLI	0000000	shamt	rs1	001	rd	0010011
XORI	imm[11:0]		rs1	100	rd	0010011
ORI	imm[11:0]		rs1	110	rd	0010011
ANDI	imm[11:0]		rs1	111	rd	0010011

上述 5 条指令为同一类型,它们都属于 I 类型的 32 位指令,低 7 位都为 7'b0010011,判断指令的具体操作类型则通过指令的第 12～14 位。

指令用法:

```
addi  rd  rs1  immediate            //立即数加法运算
slli   rd  rs1  shamt               //立即数左移运算
xori  rd  rs1  immediate            //立即数异或运算
ori   rd  rs1  immediate            //立即数或运算
andi  rd  rs1  immediate            //立即数与运算
```

除了 slli 指令以外,其余的 4 条指令是将指令中的立即数与 rs1 通用寄存器中的数据进行对应的运算,最后将运算结果保存到 rd 寄存器中。立即数在使用时需要将其有符号扩展成 32 位立即数后再进行相应的运算。slli 指令的立即数为 5 位,不需要进行扩展。

部分 R 类型指令说明如表 7-3 所示。

表 7-3　R 类型指令说明

指令	说　明					
ADD	0000000	rs2	rs1	000	rd	0110011
SLL	0000000	rs2	rs1	001	rd	0110011
XOR	0000000	rs2	rs1	100	rd	0110011
OR	0000000	rs2	rs1	110	rd	0110011
AND	0000000	rs2	rs1	111	rd	0110011

表 7-3 中 5 条指令为同一类型，它们都属于 R 类型的 32 位指令，5 条指令的低 7 位都为 7'b0110011，判断指令的具体操作类型同样通过指令的第 12～14 位。

指令用法：

```
add  rd  rs1  immediate                        //加法运算
sll   rd  rs1  immediate                       //左移运算
xor  rd  rs1  immediate                        //异或运算
or   rd  rs1  immediate                        //或运算
and  rd  rs1  immediate                        //与运算
```

5 条指令都是读取 rs1 通用寄存器中的数据与 rs2 通用寄存器中的数据进行相应的运算，最后将运算结果保存到 rd 寄存器中。

7.2.1　5 级流水介绍

5 级流水线由取指、译码、执行、访存、回写这 5 个部分组成。5 级流水中的各个阶段的具体工作如下：

(1) 取指：读取存储器中的指令，同时确定下一条指令的地址。

(2) 译码：对取出的指令进行译码，获得指令中要读取的寄存器值，操作类型或者立即数等信息，针对立即数进行有符号或者无符号扩展等操作。如果包含转移指令且满足转移条件，此阶段会给出目标地址。

(3) 执行：将译码阶段发送出来的数据根据操作类型进行运算并发送出运算结果。如果执行的指令为 Load(读)或 Store(写)指令，执行阶段也会计算读写的地址。

(4) 访存：如果有 Load(读)或 Store(写)指令，将会在此阶段访问存储器，否则此阶段可以忽略，执行阶段的数据将会直接发送到回写阶段。此阶段同时可以判断是否有异常需要处理，如果有异常则进行异常处理。

(5) 回写：将不写入存储器的运算结果写回相应的目的寄存器。

7.2.2　单条指令的 RISC-V 处理器设计

处理器设计是一个相对复杂的工作，将所有指令和功能一次性全部实现是有一定难度的。可以先从一条指令入手，再不断地丰富处理器的功能。现在将从上面提到的 ori 指令入手，开始 RISC-V 处理器的设计。各模块的完整代码存放在配套源代码的对应目录下，请到清华大学出版社官方网站本书页面下载。

ori 指令将操作数寄存器 rs1 中的整数值与 12 位立即数(进行符号位扩展)进行或(or)操作，结果写回寄存器 rd 中。

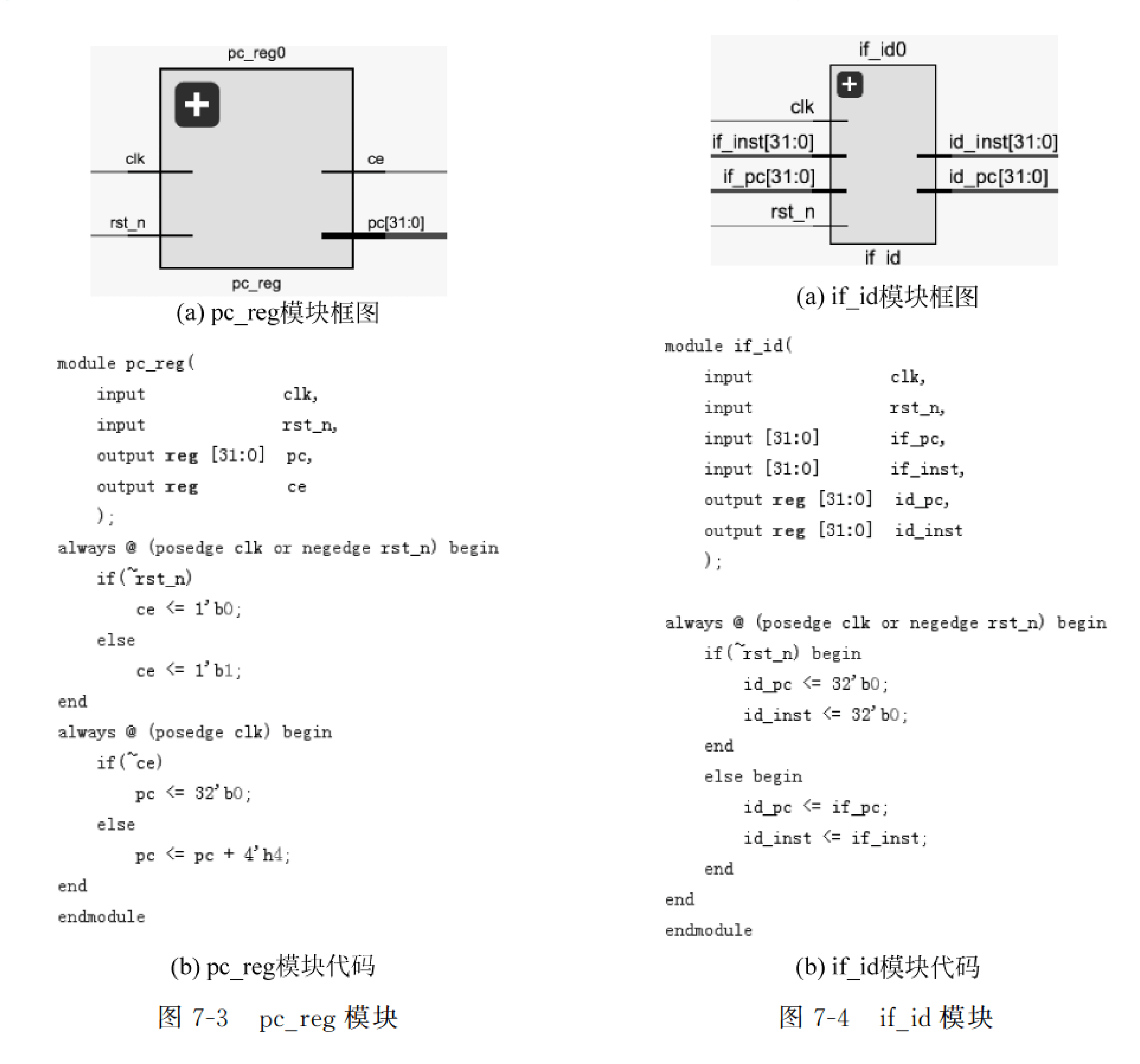
ORI	imm[11:0]	rs1	110	rd	0010011

1. 取指模块的设计

取指模块的作用是从指令存储器中取出指令，并递增 PC 值，给出下一条指令的地址，取指模块的文件名为 pc_reg.v，其框图及代码如图 7-3 所示。

当复位信号有效时，使能为零，PC 值清零；否则，使能信号为 1，PC 值每个时钟周期加 4。因为对于 32 位指令来说，一条指令对应 4 字节，PC 值加 4 意味着取下一条指令。

if_id. v 文件为取指和译码阶段之间的寄存器模块，它将取指阶段取出的指令暂存起来，等到下一个时钟周期将取指阶段的结果发送到译码阶段，其框图及代码如图 7-4 所示。



(a) pc_reg模块框图

```
module pc_reg(
    input               clk,
    input               rst_n,
    output reg [31:0]   pc,
    output reg          ce
    );
always @ (posedge clk or negedge rst_n) begin
    if(~rst_n)
        ce <= 1'b0;
    else
        ce <= 1'b1;
end
always @ (posedge clk) begin
    if(~ce)
        pc <= 32'b0;
    else
        pc <= pc + 4'h4;
end
endmodule
```

(b) pc_reg模块代码

图 7-3 pc_reg 模块

(a) if_id模块框图

```
module if_id(
    input               clk,
    input               rst_n,
    input  [31:0]       if_pc,
    input  [31:0]       if_inst,
    output reg [31:0]   id_pc,
    output reg [31:0]   id_inst
    );

always @ (posedge clk or negedge rst_n) begin
    if(~rst_n) begin
        id_pc <= 32'b0;
        id_inst <= 32'b0;
    end
    else begin
        id_pc <= if_pc;
        id_inst <= if_inst;
    end
end
endmodule
```

(b) if_id模块代码

图 7-4 if_id 模块

2. 译码模块的设计

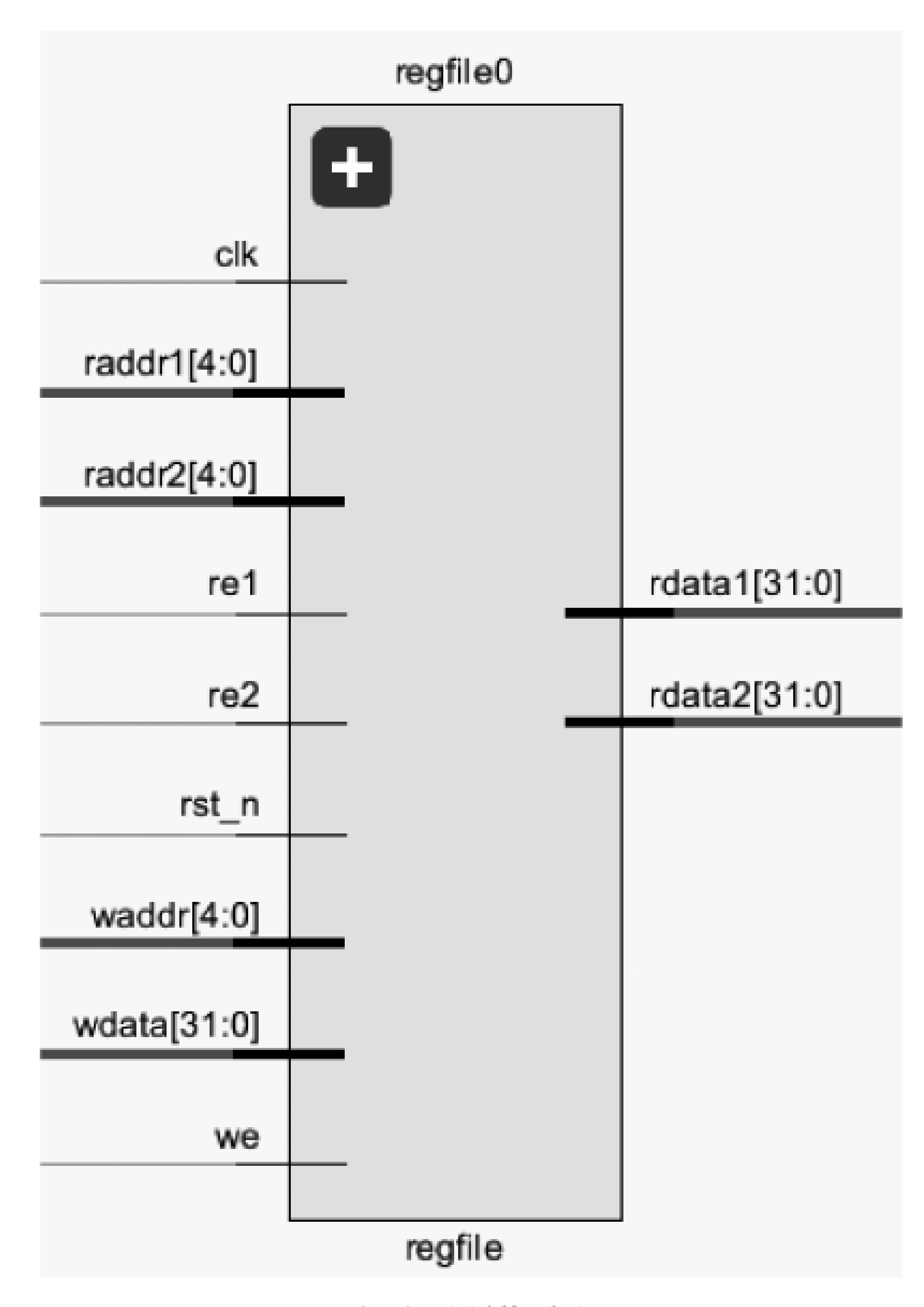
译码阶段会将取指模块发送过来的信息进行译码，得出指令中包含的操作数。运算类型和写入目的寄存器的地址等信息发送到下一级的执行阶段。译码阶段包含三个文件，它们分别是 id. v、regfile. v、id_ex. v。完整代码在配套源代码对应目录下。

regfile. v 创建了一个有 32 个 32 位的寄存器组，包含一个写端口和两个读端口。写入寄存器和读端口 1 的框图及部分代码如图 7-5 所示，读端口 2 的代码与端口 1 相似，在此不做赘述。

id. v 文件将指令的操作类型、操作数 1、操作数 2、目的寄存器地址等信息取出并发送到执行阶段，通过特定位来识别指令类型和运算方式，其框图及代码如图 7-6 所示。

首先判断指令是哪种操作类型。确定操作类型后，通过 op1 进一步确定具体为何种运算方式，根据指令信息判断是否要读操作数。对于 I 类型指令 ori，由于只需要读取操作数 1，所以读操作数 2 为 0，并对立即数进行符号位扩展。

在处理器的流水线中面临数据相关问题。数据相关是指由于相邻的两条或多条指令使用了相同的数据地址而发生的关联。这里所说的数据地址包括存储单元地址和寄存器地址。例如，有如下 3 条指令依次流入相应的流水线中：



(a) 寄存器模块框图

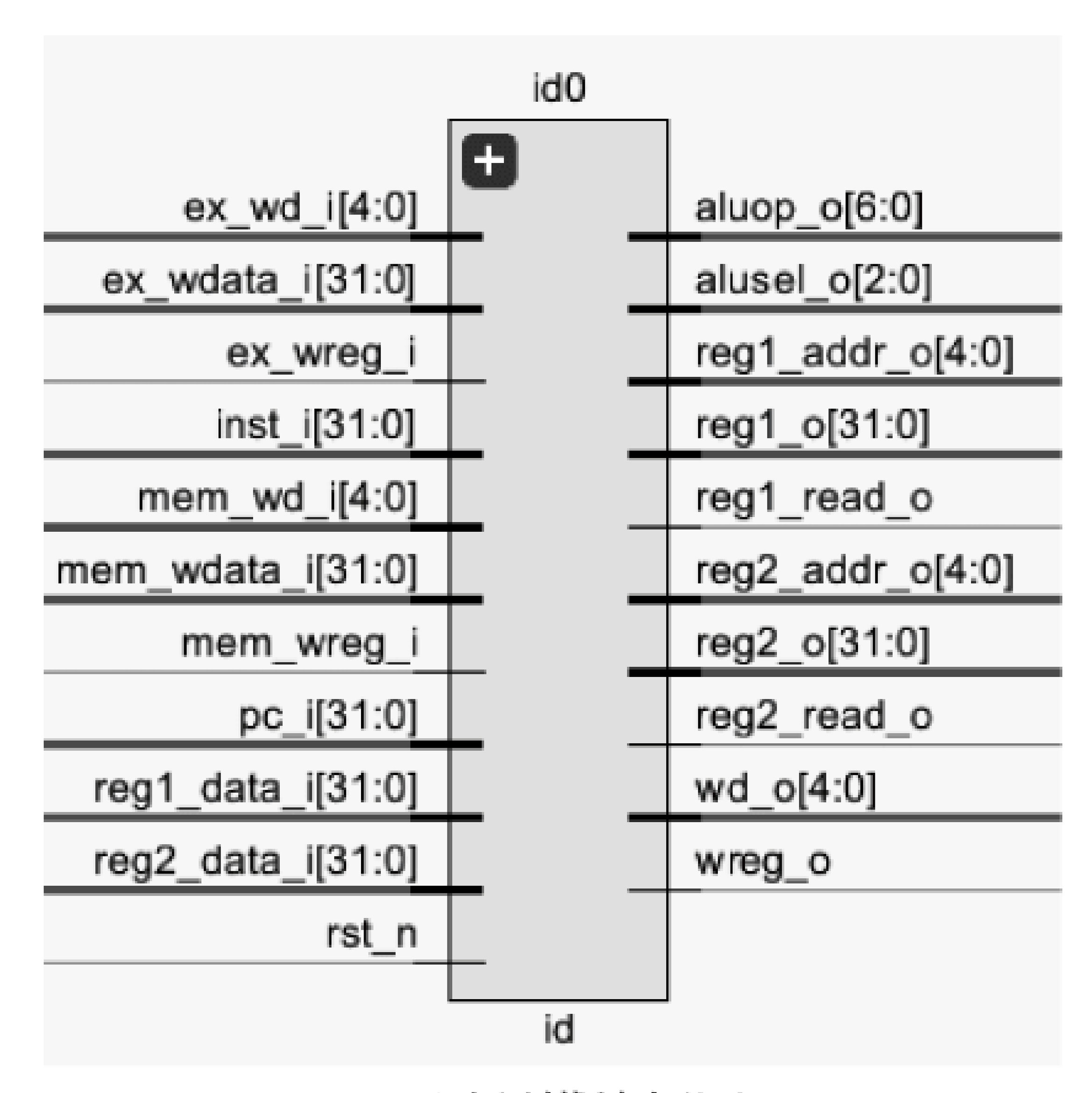
```
module regfile(
    input              rst_n,
    input              clk,
    input  [4:0]       waddr,
    input  [31:0]      wdata,
    input              we,
    input  [4:0]       raddr1,
    input              re1,
    output reg [31:0]  rdata1,
    input  [4:0]       raddr2,
    input              re2,
    output reg [31:0]  rdata2
    );
reg [31:0] mem_r [0:31];

always @ (posedge clk or negedge rst_n) begin      //写入寄存器
    if(rst_n) begin
        if((waddr != 5'b0) && (we))
            mem_r[waddr] <= wdata;
    end
end

always @ (*) begin                                  //从端口1读出数据
    if(~rst_n)
        rdata1 <= 32'b0;
    else if(raddr1 == 5'b0)
        rdata1 <= 32'b0;
    else if((raddr1 == waddr) && re1 && we)         //如果写入的地址与要读出的地址相同
        rdata1 <= wdata;                            //则直接将写入数据读出
    else if(re1)
        rdata1 <= mem_r[raddr1];
    else
        rdata1 <= 32'b0;
end
```

(b) 寄存器模块代码

图 7-5　寄存器模块



(a) 译码模块框图

```
wire [6:0] op  = inst_i[6:0];                         //运算类型
wire [2:0] op1 = inst_i[14:12];                       //具体运算方式

case(op)
    7'b0010011: begin                                   //立即数操作
        case (op1)
            3'b110: begin                               //addi,xori,ori
                wreg_o <= 1'b1;                         //是否写目的寄存器
                aluop_o <= op;                          //运算类型
                alusel_o <= op1;                        //运算方式
                reg1_read_o <= 1'b1;                    //是否读操作数1
                reg2_read_o <= 1'b0;                    //是否读操作数2
                imm <= {{20{inst_i[31]}} , inst_i[31:20]};  //立即数扩展
            end
            default: begin
            end
        endcase
    end
```

(b) 译码模块代码

图 7-6 译码模块

ADD R1,R2,R3; (R2)+(R3)→R1

SUB R1,R3,R2; (R3)−(R2)→R1

AND R2,R1,R3; (R1)&(R3)→R2

例如上述三条指令,第三条指令与前面两条指令之间存在关于寄存器 R1 的先写后读(RAW)数据相关。第三条指令应该读取的为第二条指令结束后 R1 的结果,但由于流水线的原因,在第三条指令读取 R1 时,前面指令的执行结果还没有回写到 R1 中,因此会造成计算错误。

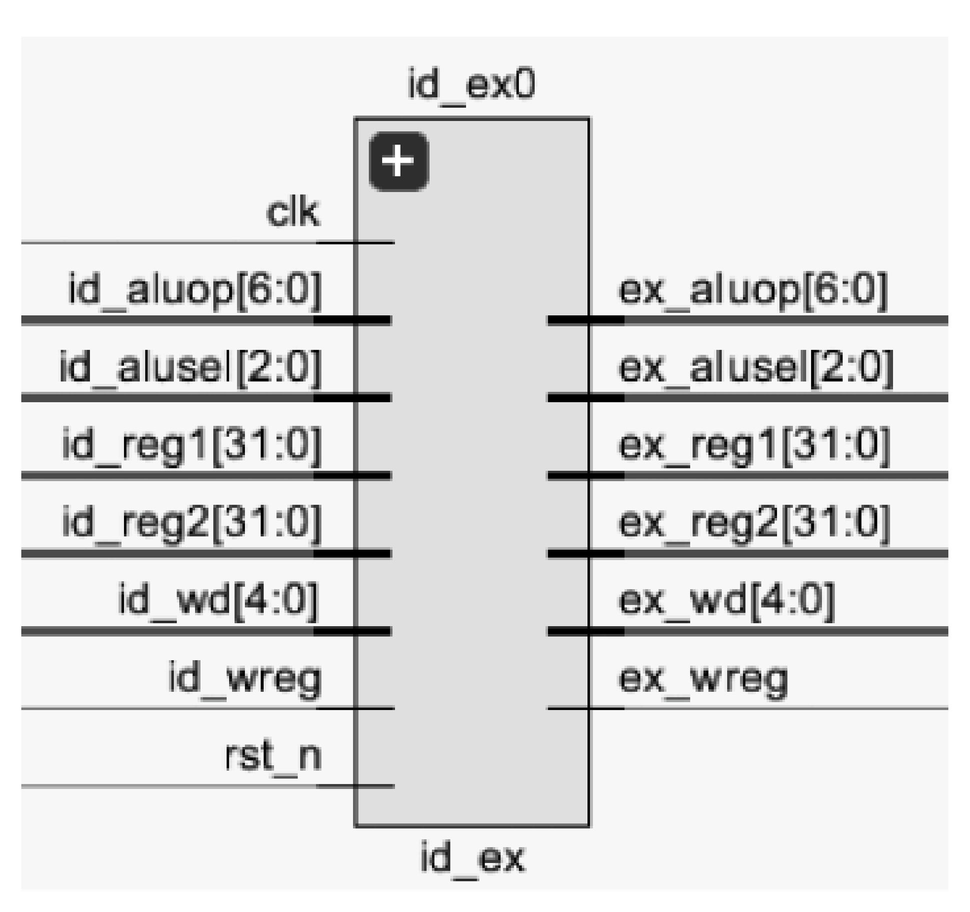
为了避免数据相关,该 RISC-V 处理器中使用了数据前推的方法,就是将计算结果从产生处直接推送到后续指令所需要的相应位置,从而避免流水线暂停。图 7-7 显示了操作数 1 防止数据相关部分的代码,如果发生数据相关则从相应位置读取所需的数据,否则直接读取寄存器中的结果给下一阶段。操作数 2 的代码以此类似,全部代码请查看配套源代码中的相应文件。

```
always @ (*) begin
    if(~rst_n)
        reg1_o <= 0;
    else if((reg1_read_o) && (ex_wreg_i) && (ex_wd_i == reg1_addr_o))    //如果执行阶段的数据为译码阶段所要
                                                                          //读取的数据,则直接将其送回译码阶段
        reg1_o <= ex_wdata_i;
    else if((reg1_read_o) && (mem_wreg_i) && (mem_wd_i == reg1_addr_o))  //如果访存阶段的数据为译码阶段所要
                                                                          //读取的数据,则直接将其送回译码阶段
        reg1_o <= mem_wdata_i;

    else if(reg1_read_o)
        reg1_o <= reg1_data_i;
    else if(~reg1_read_o)
        reg1_o <= imm;
    else
        reg1_o <= 0;
end
```

图 7-7 数据前推实现代码

id_ex. v 文件为译码与执行阶段的寄存器,它将译码阶段的结果储存起来,在下一个时钟周期发送给执行模块。其结构与取指模块的寄存器结构类似,其框图及代码如图 7-8 所示。



(a) id_ex模块框图

```
always @ (posedge clk or negedge rst_n) begin
    if(~rst_n) begin
        ex_aluop <= 0;
        ex_alusel <= 0;
        ex_reg1 <= 0;
        ex_reg2 <= 0;
        ex_wd <= 0;
        ex_wreg <= 0;
    end
    else begin
        ex_aluop <= id_aluop;
        ex_alusel <= id_alusel;
        ex_reg1 <= id_reg1;
        ex_reg2 <= id_reg2;
        ex_wd <= id_wd;
        ex_wreg <= id_wreg;
    end
end
```

(b) id_ex模块代码

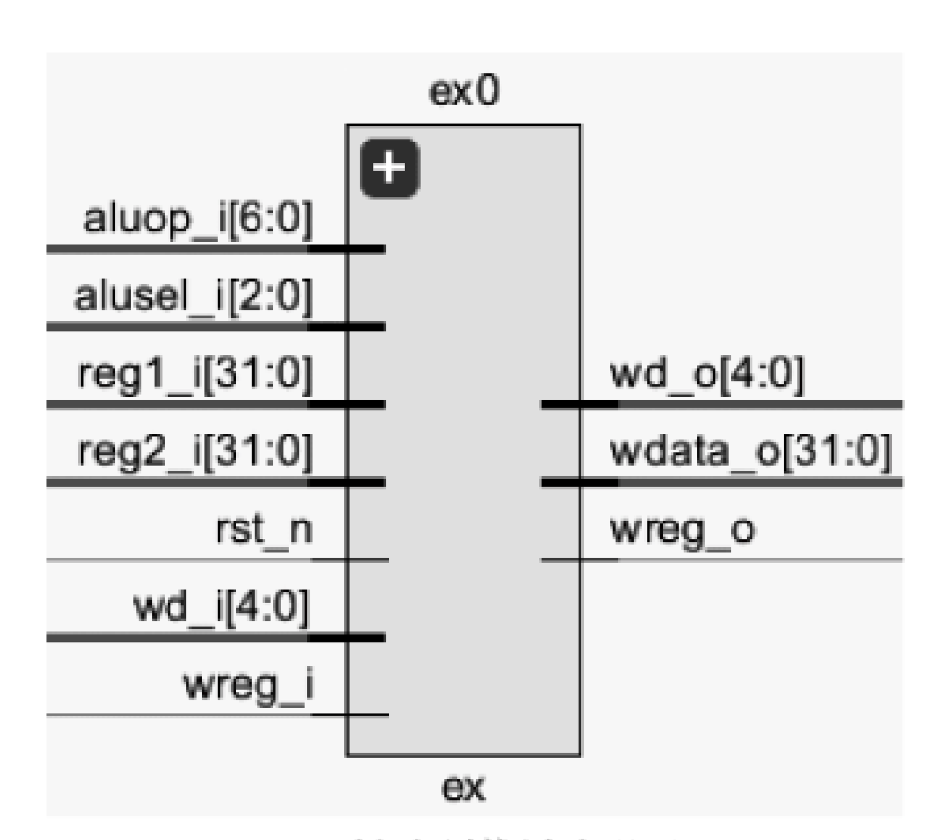
图 7-8 id_ex 模块

3. 执行模块的设计

执行模块根据之前译码模块的结果,对输入进来的操作数进行相应类型的运算。该阶

段包括 ex.v 和 ex_mem.v 两个文件。

ex.v 文件根据译码模块发送出的文件执行相应运算,其框图及代码如图 7-9 所示。



(a) 执行模块框图

```
always @ (*) begin          //通过译码阶段发送过来的信息确定具体的运算操作
    if(~rst_n)
        result <= 0;
    else begin
        case(aluop_i)
            7'b0010011: begin
                case(alusel_i)
                    3'b110: result <= reg1_i | reg2_i;    //执行相应的运算
                    default: begin
                        result <= 0;
                    end
                endcase
            end
        endcase
    end
end
```

(b) 执行模块代码

图 7-9　执行模块

将运算后的最终结果,即是否写回寄存器信号、写回目的寄存器的地址等信息发送到下一个阶段,其部分代码如图 7-10 所示。

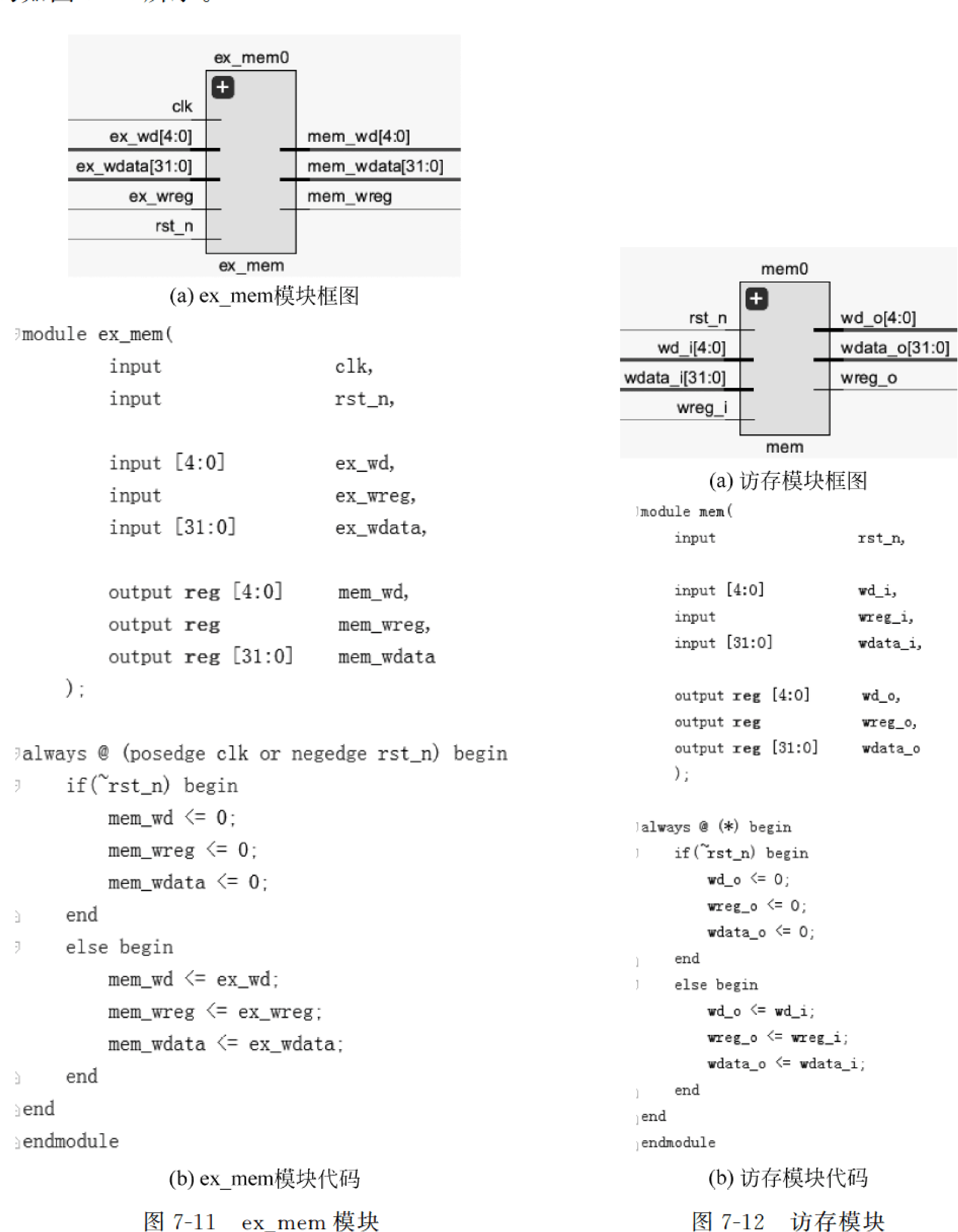
```
always @ (*) begin          //将运算结果发送到下一阶段
    wd_o <= wd_i;
    wreg_o <= wreg_i;
    case(aluop_i)
        7'b0010011: begin
            case(alusel_i)
                3'b110: wdata_o <= result;
                default: begin
                end
            endcase
        end
        default: begin
            wdata_o <= 0;
        end
    endcase
end
endmodule
```

图 7-10　判断写回部分代码

ex_mem.v 模块为执行与访存阶段之间的寄存器，它将执行阶段的结果储存起来，在下一个时钟周期发送给模块。其结构与取指和译码模块的寄存器结构类似，其框图及代码如图 7-11 所示。

4. 访存模块的设计

由于上述使用到的指令并没有涉及访问存储器的指令，所以在此模块中不执行任何操作，数据通过访存模块直接送到回写模块。在补充指令完成 RISC-V 处理器时，当涉及读写存储器或者中断异常时将会对访存模块进行相应的修改。访存模块包含 mem.v 和 mem_wb.v 两个文件。mem.v 目前只设计成一个组合逻辑电路，将输入的数据直接输出，其框图及代码如图 7-12 所示。



(a) ex_mem模块框图

```
module ex_mem(
        input               clk,
        input               rst_n,

        input [4:0]         ex_wd,
        input               ex_wreg,
        input [31:0]        ex_wdata,

        output reg [4:0]    mem_wd,
        output reg          mem_wreg,
        output reg [31:0]   mem_wdata
    );

always @ (posedge clk or negedge rst_n) begin
    if(~rst_n) begin
        mem_wd <= 0;
        mem_wreg <= 0;
        mem_wdata <= 0;
    end
    else begin
        mem_wd <= ex_wd;
        mem_wreg <= ex_wreg;
        mem_wdata <= ex_wdata;
    end
end
endmodule
```

(b) ex_mem模块代码

图 7-11　ex_mem 模块

(a) 访存模块框图

```
module mem(
    input               rst_n,

    input [4:0]         wd_i,
    input               wreg_i,
    input [31:0]        wdata_i,

    output reg [4:0]    wd_o,
    output reg          wreg_o,
    output reg [31:0]   wdata_o
    );

always @ (*) begin
    if(~rst_n) begin
        wd_o <= 0;
        wreg_o <= 0;
        wdata_o <= 0;
    end
    else begin
        wd_o <= wd_i;
        wreg_o <= wreg_i;
        wdata_o <= wdata_i;
    end
end
endmodule
```

(b) 访存模块代码

图 7-12　访存模块

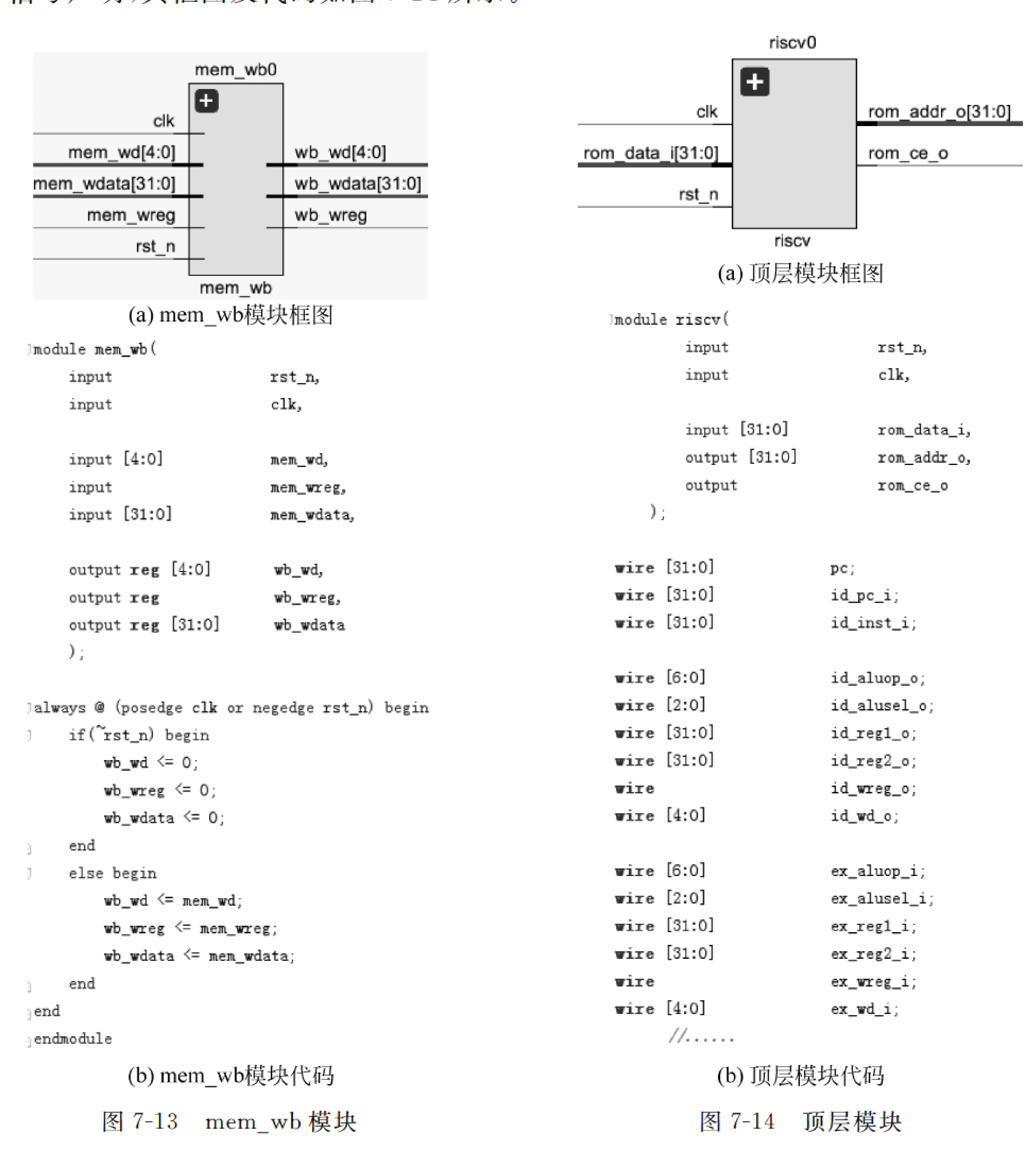
mem_wb.v 文件与之前各模块的寄存器文件类似，都是将本阶段的数据在下一个时钟周期发送到下一模块，其框图及代码如图 7-13 所示。

5. 回写模块的设计

流水线最后是将上述指令的结果写回寄存器，此部分发生在 regfile 模块，将 mem_wb.v 的输出连接到 regfile.v 模块的输入完成数据的写回。各个输入/输出的详细连接可以参考 regfile.v 和顶层模块。

6. 顶层模块的设计

顶层模块的主要目的是将上述文件中的输入/输出端口进行连接和例化，将上述各个模块连接成一个完整系统。顶层模块文件为 risc.v 文件。下面为顶层模块输入/输出定义和部分信号声明，其框图及代码如图 7-14 所示。



(a) mem_wb模块框图

```
module mem_wb(
    input                   rst_n,
    input                   clk,

    input [4:0]             mem_wd,
    input                   mem_wreg,
    input [31:0]            mem_wdata,

    output reg [4:0]        wb_wd,
    output reg              wb_wreg,
    output reg [31:0]       wb_wdata
    );

always @ (posedge clk or negedge rst_n) begin
    if(~rst_n) begin
        wb_wd <= 0;
        wb_wreg <= 0;
        wb_wdata <= 0;
    end
    else begin
        wb_wd <= mem_wd;
        wb_wreg <= mem_wreg;
        wb_wdata <= mem_wdata;
    end
end
endmodule
```

(b) mem_wb模块代码

图 7-13 mem_wb 模块

(a) 顶层模块框图

```
module riscv(
        input               rst_n,
        input               clk,

        input [31:0]        rom_data_i,
        output [31:0]       rom_addr_o,
        output              rom_ce_o
    );

wire [31:0]         pc;
wire [31:0]         id_pc_i;
wire [31:0]         id_inst_i;

wire [6:0]          id_aluop_o;
wire [2:0]          id_alusel_o;
wire [31:0]         id_reg1_o;
wire [31:0]         id_reg2_o;
wire                id_wreg_o;
wire [4:0]          id_wd_o;

wire [6:0]          ex_aluop_i;
wire [2:0]          ex_alusel_i;
wire [31:0]         ex_reg1_i;
wire [31:0]         ex_reg2_i;
wire                ex_wreg_i;
wire [4:0]          ex_wd_i;
    //......
```

(b) 顶层模块代码

图 7-14 顶层模块

对各个模块例化，将它们组合成系统，如图 7-15 所示。

到此针对 ori 指令的 5 级流水线设计已经完成了，后续内容会继续补充其内容，实现更多的功能。

7. 指令存储器 ROM 的设计

由于本处理器采用哈佛结构，即指令与数据单独存储，如果想让处理器进行功能验证必须添加对应的指令存储器，CPU 将从指令存储器中取出指令。下面介绍对于指令存储器 ROM 的设计，其框图及代码如图 7-16 所示。

```
pc_reg pc_reg0(
    .clk(clk),
    .rst_n(rst_n),
    .pc(pc),
    .ce(rom_ce_o)
);
assign rom_addr_o = pc;

if_id if_id0(
    .clk(clk),
    .rst_n(rst_n),
    .if_pc(pc),
    .if_inst(rom_data_i),
    .id_pc(id_pc_i),
    .id_inst(id_inst_i)
);
regfile regfile0(
    .clk(clk),
    .rst_n(rst_n),
    .we(wb_wreg_i),
    .waddr(wb_wd_i),
    .wdata(wb_wdata_i),
    .re1(reg1_read),
    .raddr1(reg1_addr),
    .rdata1(reg1_data),
    .re2(reg2_read),
    .raddr2(reg2_addr),
    .rdata2(reg2_data)
);
```

图 7-15　各模块例化

inst_rom0

addr[31:0]　ce　inst[31:0]

inst_rom

(a) 存储器模块框图

```
module inst_rom(
    input              ce,
    input  [31:0]      addr,
    output reg [31:0]  inst
    );

reg [31:0] inst_mem [0:255];

initial $readmemh ("inst_rom.data", inst_mem);

always @ (*) begin
    if(~ce)
        inst <= 0;
    else
        inst <= inst_mem[addr[9:2]];
end
endmodule
```

(b) 存储器模块代码

图 7-16　存储器模块

设计中使用到了 initial 语句，initial 语句只执行一次，一般只用于仿真，不能被综合。函数 $readmemh 是一条系统函数，它可以用来读取文件，但要保证文件中的指令为十六进制。inst_rom.data 文件中存储着测试 CPU 所要用到的 32 位指令，通过 $readmemh 函数可以将文件中的指令读取到 inst_mem 数组中。

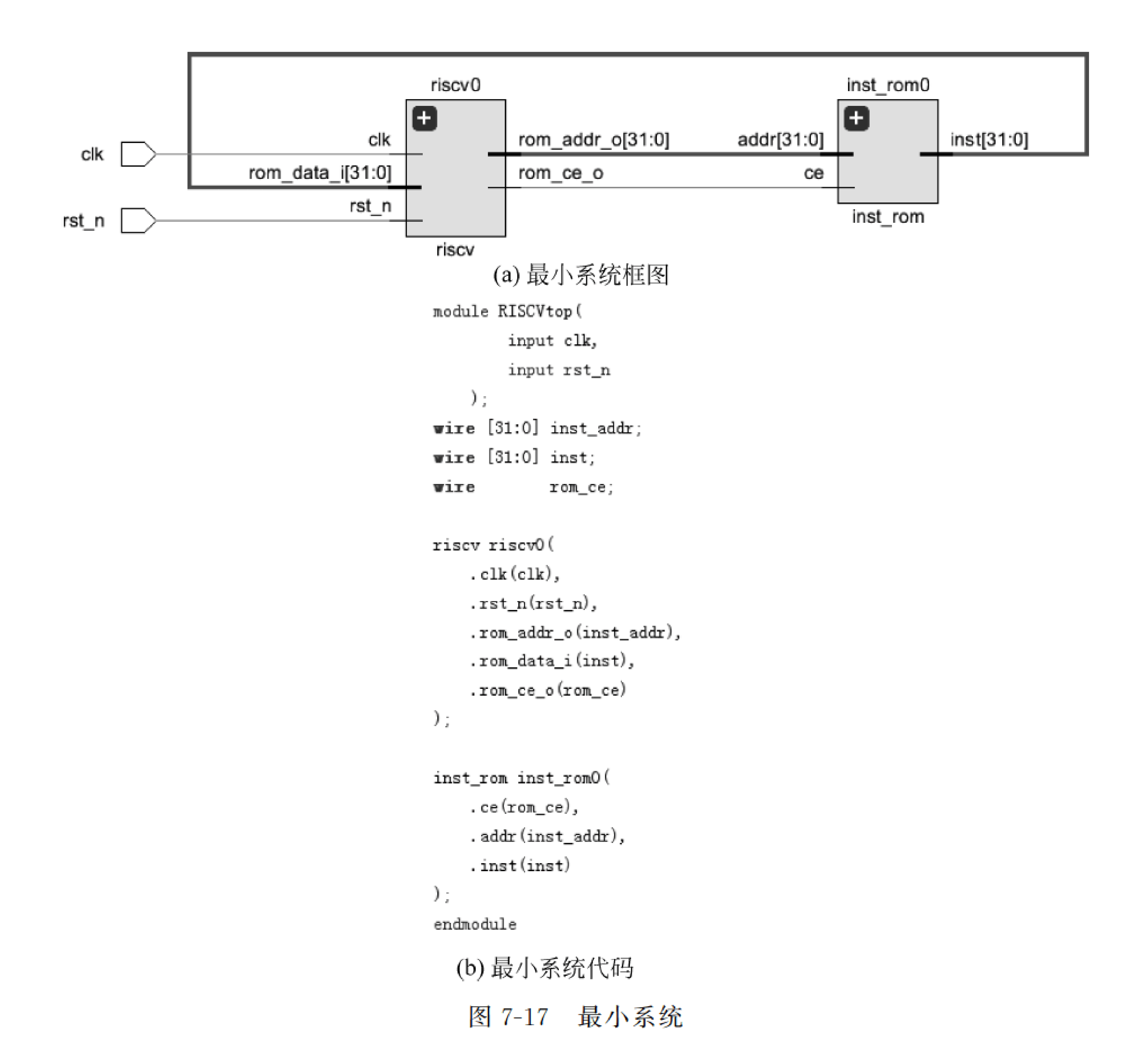
由于设计中采用字节寻址，所以每一条 32 位指令地址都要占用 4 个地址位，指令地址都要为 4 的倍数，可以将指令地址左移两位来保证地址的正确性。

8. 最小系统的设计

该模块是将指令存储器 ROM 与之前所设计的 CPU 进行连接构成一个 CPU 的最小系统。其设计方法与之前 CPU 顶层模块的设计方法相似，要将两个模块进行连线和例化。当 CPU 最小系统运行时 CPU 模块会将地址发送到指令存储器 ROM 中，指令存储器将所对应的指令读取到 CPU 中供 CPU 流水线进行执行，其框图及代码如图 7-17 所示。

9. 测试程序编写

如果想要对 CPU 最小系统进行仿真验证，还需要对指令存储器中提到的 inst_rom.data 进行编辑和编写 CPU 测试文件。对于目前单条指令的测试指令，读者可以使用配套源代码指定目录中的文件进行测试，也可以根据 RISC-V 指令集自行编写测试指令。下面



(a) 最小系统框图

```
module RISCVtop(
        input clk,
        input rst_n
    );
wire [31:0] inst_addr;
wire [31:0] inst;
wire        rom_ce;

riscv riscv0(
    .clk(clk),
    .rst_n(rst_n),
    .rom_addr_o(inst_addr),
    .rom_data_i(inst),
    .rom_ce_o(rom_ce)
);

inst_rom inst_rom0(
    .ce(rom_ce),
    .addr(inst_addr),
    .inst(inst)
);
endmodule
```

(b) 最小系统代码

图 7-17　最小系统

简单介绍一下测试指令编写的方法，如表 7-4 所示。

表 7-4　编写测试指令

测试指令	ORI S1,S0,0x2			
imm[11:0]	rs1	110	rd	0010011
000000000010	00000	110	00001	0010011
inst_rom. data	00206093			

用户首先设计对应的汇编指令，根据 RISC-V 指令集的编码规则将汇编指令转换成相应的 RISC-V 指令形式，最后将 32 位指令转换成十六进制形式写入 inst_rom. data 文件中。本书附带的测试程序为如下几条指令，其测试文件可在配套源代码相应目录下查看。

```
ORI   S1,S0,0x2          00206093
ORI   S2,S0,0x3          00306113
ORI   S3,S0,0x5          00506193
```

测试文件的编写：测试模块主要将时钟和复位信号提供给 CPU，将最小系统与测试模块进行连接，并观察最终的测试波形是否符合之前的设计要求。图 7-18 为测试模块 test. v

部分的代码。

通过 Vivado 软件自带的仿真工具进行仿真，观察波形验证是否与设计思想一致。由于测试文件中只有 ori 指令运算，所以我们可以直接观察写回寄存器的数值是否与想要得到的测试结果一致即可，在仿真界面添加 mem_r 信号来观察最终的结果，如图 7-19 所示。

```
module test(
    );
reg clk;
reg rst_n;

initial begin
    clk = 0;
    rst_n = 0;
    #100;
    rst_n = 1'b1;
end
always #20 clk = ~clk;

RISCVtop RISCVtop0(
    .clk(clk),
    .rst_n(rst_n)
);

endmodule
```

图 7-18　测试模块代码

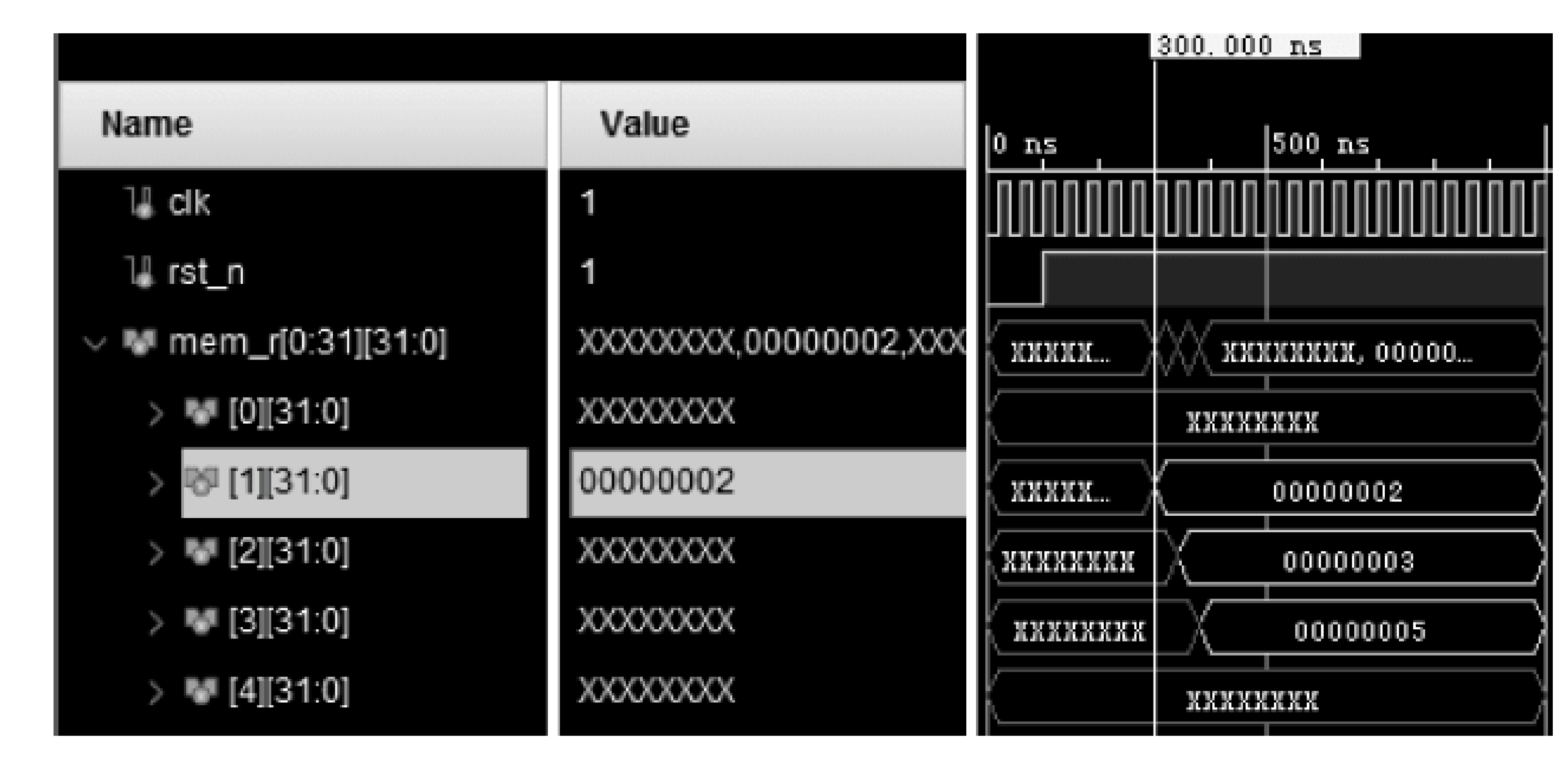


图 7-19　单条指令仿真结果

从图 7-19 中可以发现上述 3 条测试指令的运算结果全部正确，也可以看出流水线将结果逐条写回寄存器中，实现了流水的功能。到目前为止，基于 ori 指令的 RISC-V 处理器的设计就已经完全结束了。读者可以根据 RISC-V 指令集丰富该款 CPU 让其实现更多的功能。

7.2.3　2 条指令的 RISC-V 处理器设计

完成了单条指令的处理器设计后，下面在该设计的基础上添加一条指令，扩充流水线的功能。下面在处理器上添加 addi 指令。addi 指令将操作数寄存器 rs1 中的整数值与 12 位立即数(进行符号位扩展)进行加法操作，结果写回寄存器 rd 中。

从前文的指令介绍中可以发现，addi 与 ori 指令都属于 I 类型的 32 位指令，两条指令的低 7 位都为 7'b0010011，只有指令的第 12～14 位稍有不同，如表 7-5 所示。

表 7-5　2 条指令说明

指令	说　明				
ADDI	imm[11:0]	rs1	000	rd	0010011
ORI	imm[11:0]	rs1	110	rd	0010011

因此大多数模块不需要修改，只要修改设计的译码和执行两个模块，在这两个模块中添加 addi 指令相应的译码和执行代码即可完成 addi 指令的功能。

1. 译码模块的修改

由于 addi 指令和 ori 指令的 op1 部分相同，两条指令都要写回目的寄存器，读取操作数 1 并且都需要立即数而不读取寄存器 2。因此两条指令的译码方式完全一致，所以添加 addi 指令的第 12～14 位到 ori 指令的译码方式中，其他地方不做改变，修改后如图 7-20 所示。

```
case(op)
    7'b0010011: begin                              //立即数操作
        case (op1)
            3'b000,3'b110: begin                   //addi,ori
                wreg_o <= 1'b1;                    //是否写目的寄存器
                aluop_o <= op;                     //运算类型
                alusel_o <= op1;                   //运算方式
                reg1_read_o <= 1'b1;               //是否读操作数1
                reg2_read_o <= 1'b0;               //是否读操作数2
                imm <= {{20{inst_i[31]}} , inst_i[31:20]};     //立即数扩展
            end
            default: begin
            end
        endcase
    end
    default: begin
    end
endcase
```

图 7-20　译码模块修改

2. 执行模块的修改

在执行模块添加 addi 指令的执行语句，由于它与 ori 指令的运算类型都为立即数运算，所以在 ori 指令的预算类型中添加 addi 的执行语句，如图 7-21 所示。

```
always @ (*) begin          //通过译码阶段发送过来的信息确定具体的运算操作
    if(~rst_n)
        result <= 0;
    else begin
        case(aluop_i)
            7'b0010011: begin
                case(alusel_i)
                    3'b000: result <= reg1_i + reg2_i;   //执行相应的运算
                    3'b110: result <= reg1_i | reg2_i;
                    default: begin
                        result <= 0;
                    end
                endcase
            end
            default: begin
            end
        endcase
    end
end
```

图 7-21　添加执行语句

将 addi 指令的运算结果添加到与 ori 指令相同的数据输出端口，如图 7-22 所示。

3. 测试程序编写

修改译码和执行模块后已经成功完成了 addi 功能的添加，下面修改测试文件以验证设计的正确性。在测试文件中添加一条 addi 指令，读者可以使用配套源代码指定目录中的文件进行测试，也可以根据上文提到的测试代码编写的方法自行编写测试指令。本书附带的

```
always @ (*) begin            //将运算结果发送到下一阶段
    wd_o <= wd_i;
    wreg_o <= wreg_i;
    case(aluop_i)
        7'b0010011: begin
            case(alusel_i)
                3'b000,3'b110: wdata_o <= result;
                default: begin
                end
            endcase
        end
        default: begin
            wdata_o <= 0;
        end
    endcase
end
```

图 7-22 添加输出

测试程序为如下指令,本测试文件可在配套源代码相应目录下查看。addi 指令的测试代码是在 ori 指令测试代码的基础上添加了一条 addi 指令来验证其功能。

```
ORI   S1,S0,0x2          00206093
ORI   S2,S0,0x3          00306113
ORI   S3,S0,0x5          00506193
ADDI  S4,S1,0x1          00108213
```

从图 7-23 中可以发现上述测试指令的运算结果全部正确,由图也可以看出流水线将结果逐条写回寄存器中,实现了流水的功能。到目前为止基于两条指令的 RISC-V 处理器的设计就已经完全结束了,读者可以根据 RISC-V 指令集丰富该款 CPU 让其实现更多的功能。

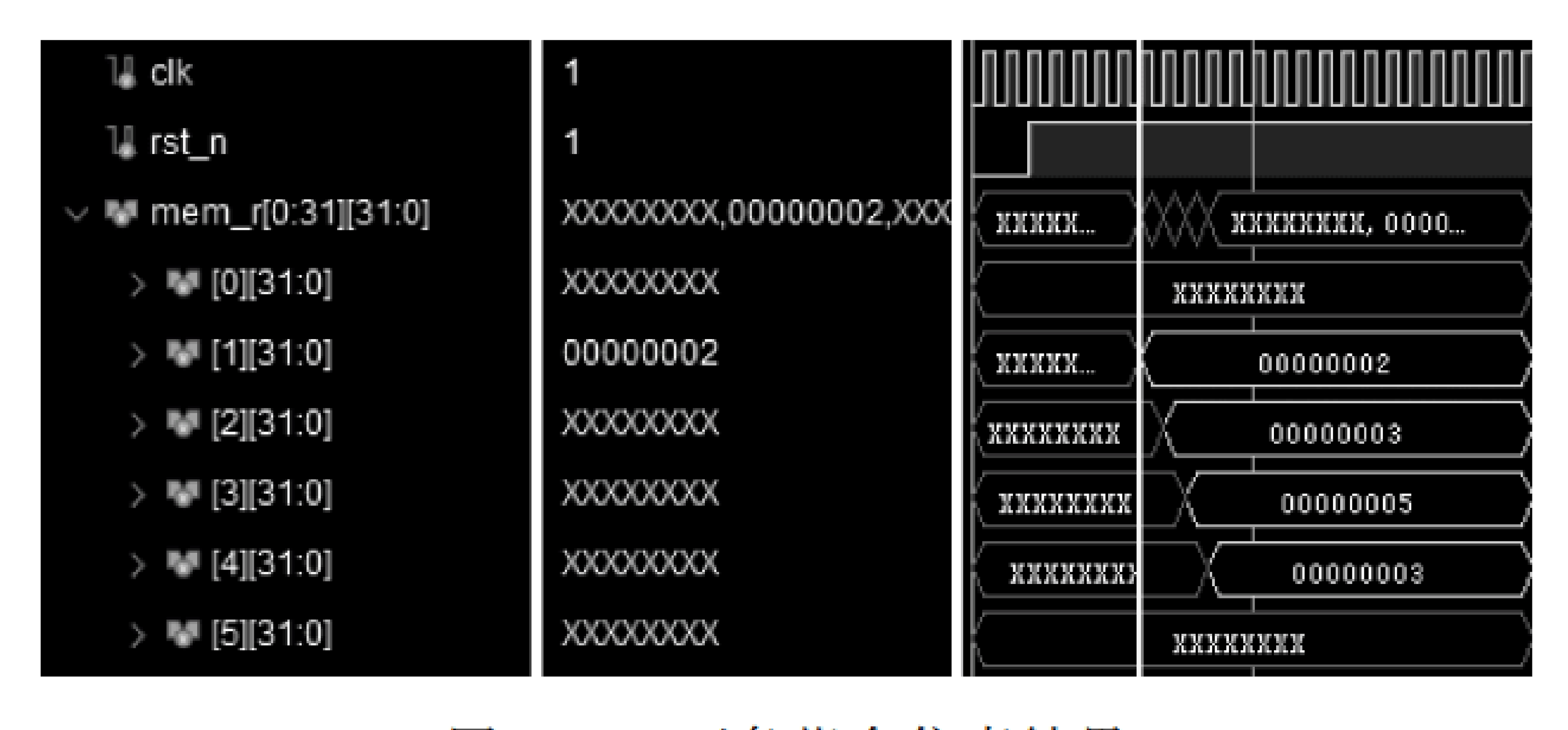


图 7-23 两条指令仿真结果

7.2.4 3 条指令的 RISC-V 处理器设计

上文介绍的两条指令都属于 I 类型的 32 位指令,如果添加其他类型的指令处理器将如何修改呢?下面将在处理器中添加一条 R 类型的 32 位指令,该种指令的低 7 位都为 7'b0110011,R 指令都是读取 rs1 通用寄存器中的数据与 rs2 通用寄存器中的数据进行相应的运算,最后将运算结果保存到 rd 寄存器中。表 7-6 为 ADDI 与 ADD 二指令的说明。

表 7-6 ADDI 与 ADD 二指令的说明

指令	说明					
ADDI	imm[11:0]		rs1	000	rd	0010011
ADD	0000000	rs2	rs1	000	rd	0110011

通过观察发现 R 类型的 add 指令和 I 类型的 addi 指令相比其低 7 位不同，并且前者需要读取寄存器 2 的数据。虽然相比 7.2.3 节两条指令的差异有所增大，但通过修改译码模块和执行模块同样可以实现相应的功能。

1. 译码模块的修改

由于 add 指令的低 7 位与之前两条指令的低 7 位不同，所以在译码阶段增加针对 R 指令类型的译码方式，如图 7-24 所示。

```
case(op)
    7'b0010011: begin                              //立即数操作
        case (op1)
            3'b000,3'b110: begin                   //addi,ori
                wreg_o <= 1'b1;                    //是否写目的寄存器
                aluop_o <= op;                     //运算类型
                alusel_o <= op1;                   //运算方式
                reg1_read_o <= 1'b1;               //是否读操作数1
                reg2_read_o <= 1'b0;               //是否读操作数2
                imm <= {{20{inst_i[31]}} , inst_i[31:20]};    //立即数扩展
            end
            default: begin
            end
        endcase
    end
    7'b0110011: begin                              //运算操作
        case(op1)
            3'b000: begin                          //add
                wreg_o <= 1'b1;
                aluop_o <= op;
                alusel_o <= op1;
                reg1_read_o <= 1'b1;
                reg2_read_o <= 1'b1;
            end
            default: begin
            end
        endcase
    end
```

图 7-24 增加译码方式

add 指令与前文的两条 I 类型指令最大的区别为需要读取操作数 2，所以读操作数 2 的值为 1，并且不需要立即数。

2. 执行模块的修改

与 addi 指令添加执行语句的情况类似，但由于其低 7 位与 I 指令不同，所以单独增加 R 类型的执行语句，如图 7-25 所示。

将 add 指令的运算结果添加到数据输出端口，如图 7-26 所示。

```
always @ (*) begin          //通过译码阶段发送过来的信息确定具体的运算操作
    if(~rst_n)
        result <= 0;
    else begin
        case(aluop_i)
            7'b0010011: begin
                case(alusel_i)
                    3'b000: result <= reg1_i + reg2_i;  //执行相应的运算
                    3'b110: result <= reg1_i | reg2_i;
                    default: begin
                        result <= 0;
                    end
                endcase
            end
            7'b0110011: begin
                case(alusel_i)
                    3'b000: result <= reg1_i + reg2_i;
                endcase
            end
            default: begin
            end
        endcase
    end
end
```

图 7-25　增加执行语句

```
always @ (*) begin          //将运算结果发送到下一阶段
    wd_o <= wd_i;
    wreg_o <= wreg_i;
    case(aluop_i)
        7'b0010011: begin
            case(alusel_i)
                3'b000,3'b110: wdata_o <= result;
                default: begin
                end
            endcase
        end
        7'b0110011: begin
            case(alusel_i)
                3'b000: wdata_o <= result;
                default: begin
                end
            endcase
        end
        default: begin
            wdata_o <= 0;
        end
    endcase
end
```

图 7-26　增加结果输出

3. 测试程序编写

修改译码和执行模块后已经成功完成了 add 功能的添加，下面修改测试文件来验证设计的正确性。在测试文件中添加一条 add 指令，读者可以使用配套源代码指定目录中的文

件进行测试，也可以根据上文提到的测试代码编写的方法自行编写测试指令。本书附带的测试程序为如下指令，本测试文件可在配套源代码相应目录下查看。add 指令的测试代码是在之前两条指令测试代码的基础上添加了一条 add 指令来验证其功能。

```
ORI   S1,S0,0x2         00206093
ORI   S2,S0,0x3         00306113
ORI   S3,S0,0x5         00506193
ADDI  S4,S1,0x1         00108213
ADD   S5,S1,S2          002082b3
```

从图 7-27 中可以发现上述测试指令的运算结果全部正确，由图也可以看出流水线将结果逐条写回寄存器中，实现了流水的功能。到目前为止基于三条指令的 RISC-V 处理器的设计就已经完全结束了，读者可以根据 RISC-V 指令集丰富该款 CPU 让其实现更多的功能。

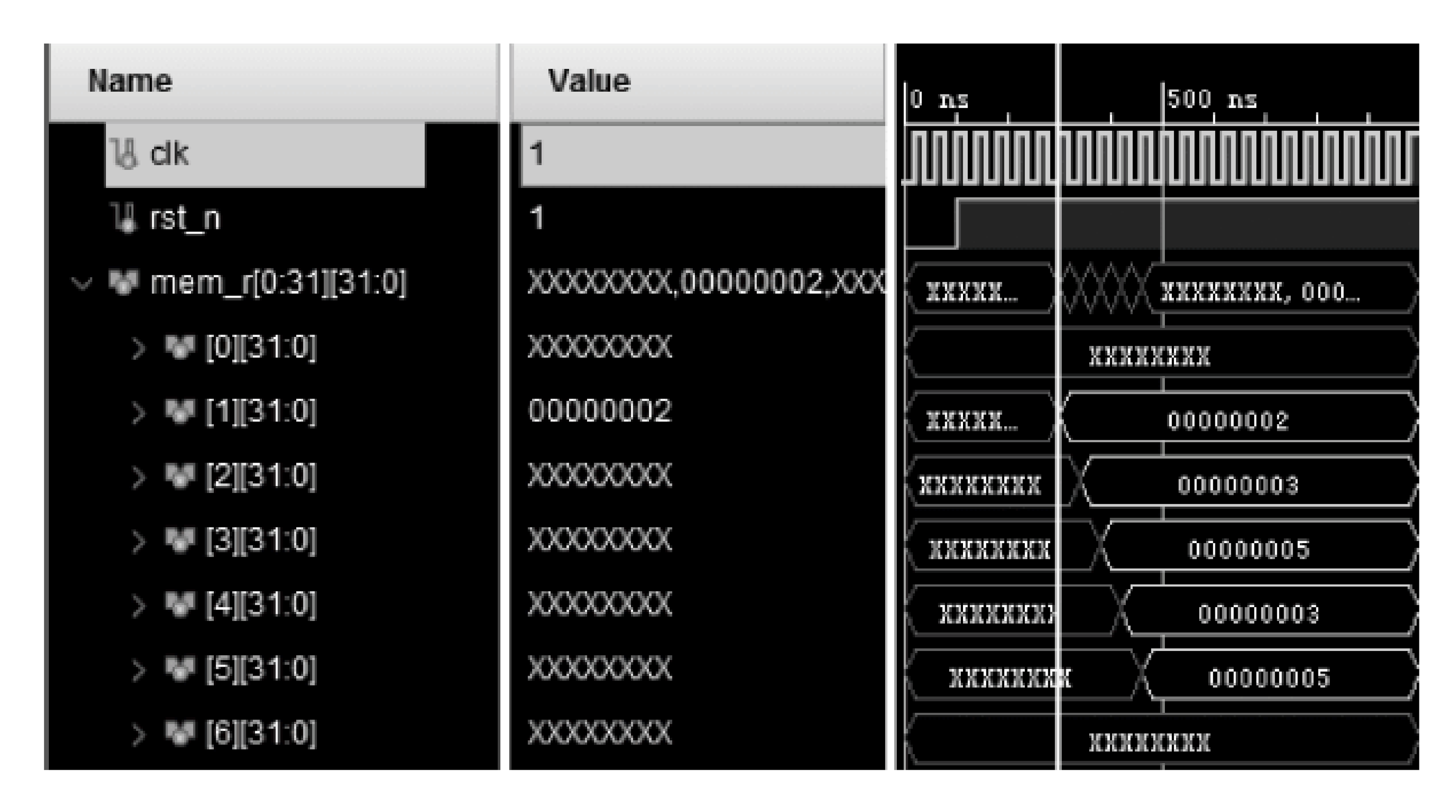


图 7-27　三条指令仿真结果

7.2.5　10 条指令的 RISC-V 处理器设计

通过上两节对于添加指令的介绍，相信读者对于指令的增加有了一定的了解。下面来实现 10 条指令的 RISC-V 处理器的设计，由于这 10 条指令都为 I 类型或 R 类型的指令，所以添加指令时只需要改变译码模块和执行模块即可。对于其他类型的指令读者也可以模仿添加 R 类型指令时的修改方法自行扩充。

10 条指令说明，如表 7-7 所示。

表 7-7　10 条指令说明

指令	说明					
ADDI	imm[11:0]		rs1	000	rd	0010011
SLLI	**0000000**	**shamt**	**rs1**	**001**	**rd**	**0010011**
XORI	imm[11:0]		rs1	100	rd	0010011
ORI	imm[11:0]		rs1	110	rd	0010011
ANDI	imm[11:0]		rs1	111	rd	0010011
ADD	0000000	rs2	rs1	000	rd	0110011
SLL	0000000	rs2	rs1	001	rd	0110011
XOR	0000000	rs2	rs1	100	rd	0110011
OR	0000000	rs2	rs1	110	rd	0110011
AND	0000000	rs2	rs1	111	rd	0110011

上述 10 条指令在添加时都遵循之前两节的指令添加方式，唯一需要注意的为 slli 指令。slli 指令对操作数寄存器 rs1 中的整数值进行逻辑左移，其移位量是 5 位立即数，结果写回到寄存器 rd 中。因此 slli 指令的立即数不需要进行符号位的扩展，在修改译码模块时要对它单独进行添加。

1. 译码模块的修改

将其他几条指令的第 12～14 位添加到对应指令类型中。由于 slli 指令的立即数不需要进行符号位的扩展所以单独对其进行译码，如图 7-28 所示。

```
case(op)
    7'b0010011: begin                              //立即数操作
        case (op1)
            3'b000,3'b100,3'b110,3'b111: begin     //addi, xori, ori, andi
                wreg_o <= 1'b1;                    //是否写目的寄存器
                aluop_o <= op;                     //运算类型
                alusel_o <= op1;                   //运算方式
                reg1_read_o <= 1'b1;               //是否读操作数1
                reg2_read_o <= 1'b0;               //是否读操作数2
                imm <= {{20{inst_i[31]}} , inst_i[31:20]};  //立即数扩展
            end
            3'b001: begin                          //slli
                wreg_o <= 1'b1;                    //是否写目的寄存器
                aluop_o <= op;                     //运算类型
                alusel_o <= op1;                   //运算方式
                reg1_read_o <= 1'b1;               //是否读操作数1
                reg2_read_o <= 1'b0;               //是否读操作数2
                imm <= inst_i[24:20];              //移位量
            end
            default: begin
            end
        endcase
    end
```

图 7-28　添加 slli 指令

对 5 条 I 类型的指令添加完成后，继续添加 R 类型指令，由于 R 类型的指令基本一致，可以直接添加，如图 7-29 所示。

```
7'b0110011: begin                                  //运算操作
    case(op1)
        3'b000,3'b001,3'b100,3'b110,3'b111: begin  //add, sll, xor, or, and
            wreg_o <= 1'b1;
            aluop_o <= op;
            alusel_o <= op1;
            reg1_read_o <= 1'b1;
            reg2_read_o <= 1'b1;
        end
        default: begin
        end
    endcase
end
default: begin
end
```

图 7-29　添加 R 类型指令

2. 执行模块的修改

执行模块的修改与上两节所用到的方法基本相同，将相应类型的执行语句添加到对应类型语句中，如图 7-30 所示。

```
case(aluop_i)
    7'b0010011: begin
        case(alusel_i)
            3'b000: result <= reg1_i + reg2_i;  //执行相应的运算
            3'b001: result <= reg1_i << reg2_i;
            3'b100: result <= reg1_i ^ reg2_i;
            3'b110: result <= reg1_i | reg2_i;
            3'b111: result <= reg1_i & reg2_i;
            default: begin
                result <= 0;
            end
        endcase
    end
    7'b0110011: begin
        case(alusel_i)
            3'b000: result <= reg1_i + reg2_i;
            3'b001: result <= reg1_i << reg2_i;
            3'b100: result <= reg1_i ^ reg2_i;
            3'b110: result <= reg1_i | reg2_i;
            3'b111: result <= reg1_i & reg2_i;
        endcase
    end
    default: begin
    end
endcase
```

图 7-30 执行模块修改

3. 测试程序编写

修改译码和执行模块后已经成功完成了 10 条指令的添加，下面修改测试文件来验证设计的正确性。在测试文件中添加指令，读者可以使用配套源代码指定目录中的文件进行测试，也可以根据上文提到的测试代码编写的方法自行编写测试指令。本书附带的测试程序为如下指令，本测试文件可在配套源代码相应目录下查看。增加指令的测试代码是在之前三条指令测试代码的基础上添加了两条指令来验证其功能。

```
ORI   S1,S0,0x2          00206093
ORI   S2,S0,0x3          00306113
ORI   S3,S0,0x5          00506193
ADDI  S4,S1,0x1          00108213
ADD   S5,S1,S2           002082b3
XOR   S6,S2,S3           00314333
SLLI  S7,S1,S2           00209393
AND   S7,S7,S3           0033f3b3
```

从图 7-31 中可以发现上述测试指令的运算结果全部正确，由图也可以看出流水线将结果逐条写回寄存器中，实现了流水的功能。到目前为止基于 10 条指令的 RISC-V 处理器的

设计就已经完全结束了，读者可以根据 RISC-V 指令集丰富该款 CPU，让其实现更多的功能。

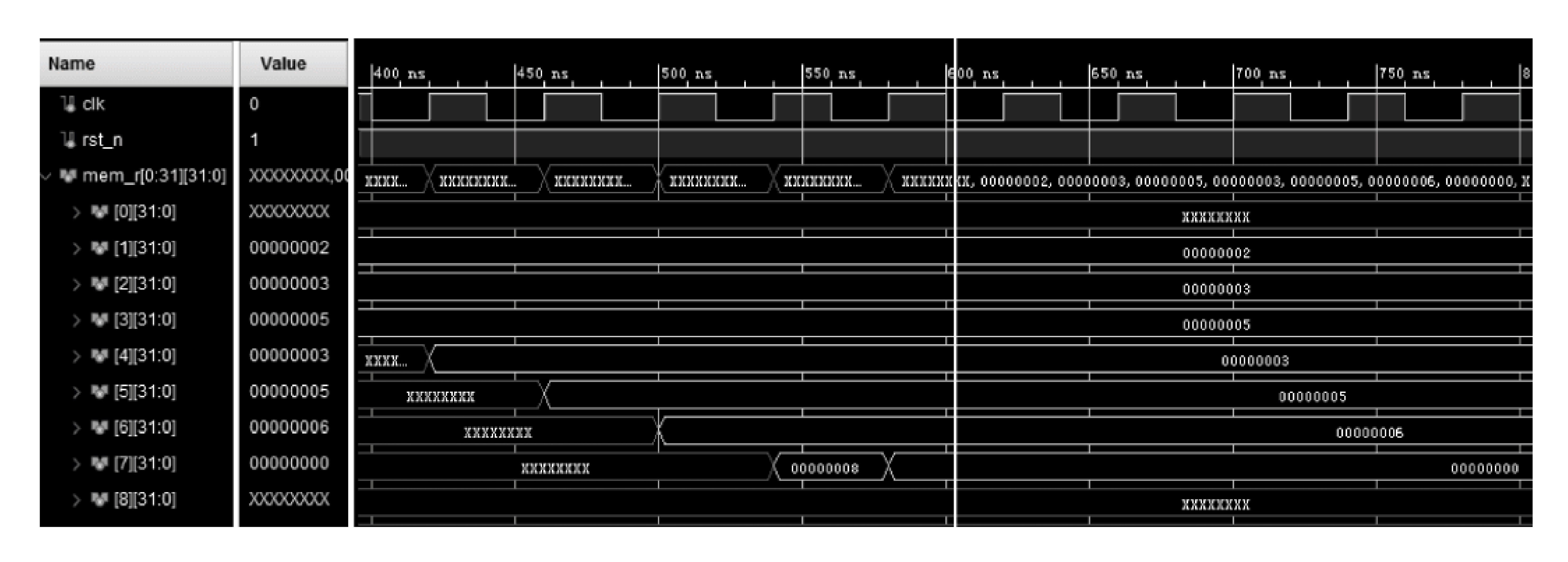


图 7-31　10 条指令仿真结果

7.3　开源 RISC-V 处理器蜂鸟 E200 介绍

蜂鸟 E200 系列处理器目标是打造一款世界上最小面积与最高能效比的 RISC 处理器。其开源代码可以在著名开源网站 GitHub 上搜索"e200_opensource"查找到，蜂鸟 E200 系列处理器核的特性简介如下：

(1) E200 系列处理器核采用 2 级流水线结构，通过一流的处理器架构设计，该 CPU 核的功耗与面积均优于同级 ARM Cortex-M0＋核，实现业界最高的能效比与最低的成本。

(2) E200 系列处理器核能够运行 RISC-V 指令集，支持 RV32I/E/A/M/C/F/D 等指令子集的配置组合，支持机器模式。

(3) E200 系列处理器核提供标准的 JTAG 调试接口以及成熟的软件调试工具。

(4) E200 系列处理器核提供成熟的 GCC 编译工具链，以及 Linux 与 Windows 图形化软件开发工具。

(5) E200 系列处理器核配套 SoC 提供紧耦合系统 IP 模块，包括中断控制器、计时器、UART、QSPI、PWM 等，以及 Ready-to-Use 的 SoC 平台与 FPGA 原型系统。

蜂鸟 E200 主要面向极低功耗与极小面积的场景，目标是替代传统的 8051 内核或者 Cortex-M 系列内核应用于 IoT 或其他低功耗场景。同时蜂鸟 E200 作为结构精简的处理器核，可谓"蜂鸟虽小、五脏俱全"。源代码全部开源公、文档翔实，适合作为在校师生学习 RISC-V 处理器设计（使用 Verilog 语言）的教学实验或自学案例。蜂鸟 E200 系列处理器的系统示意图如图 7-32 所示。

表 7-8 所列，蜂鸟 E203 核心的功耗面积和性能均优于 ARM 的 Cortex-M0＋处理器核（M0＋是 ARM 最小面积的处理器核心）；蜂鸟 E205 核心的功耗面积和性能均优于 Cortex-M3 处理器核。而 E205fd 更提供了目前仅在 Cortex-M7 中才具备的双精度浮点特性（Cortex-M7 是面积特别大的处理器核，难以适应极低功耗要求的场景）。

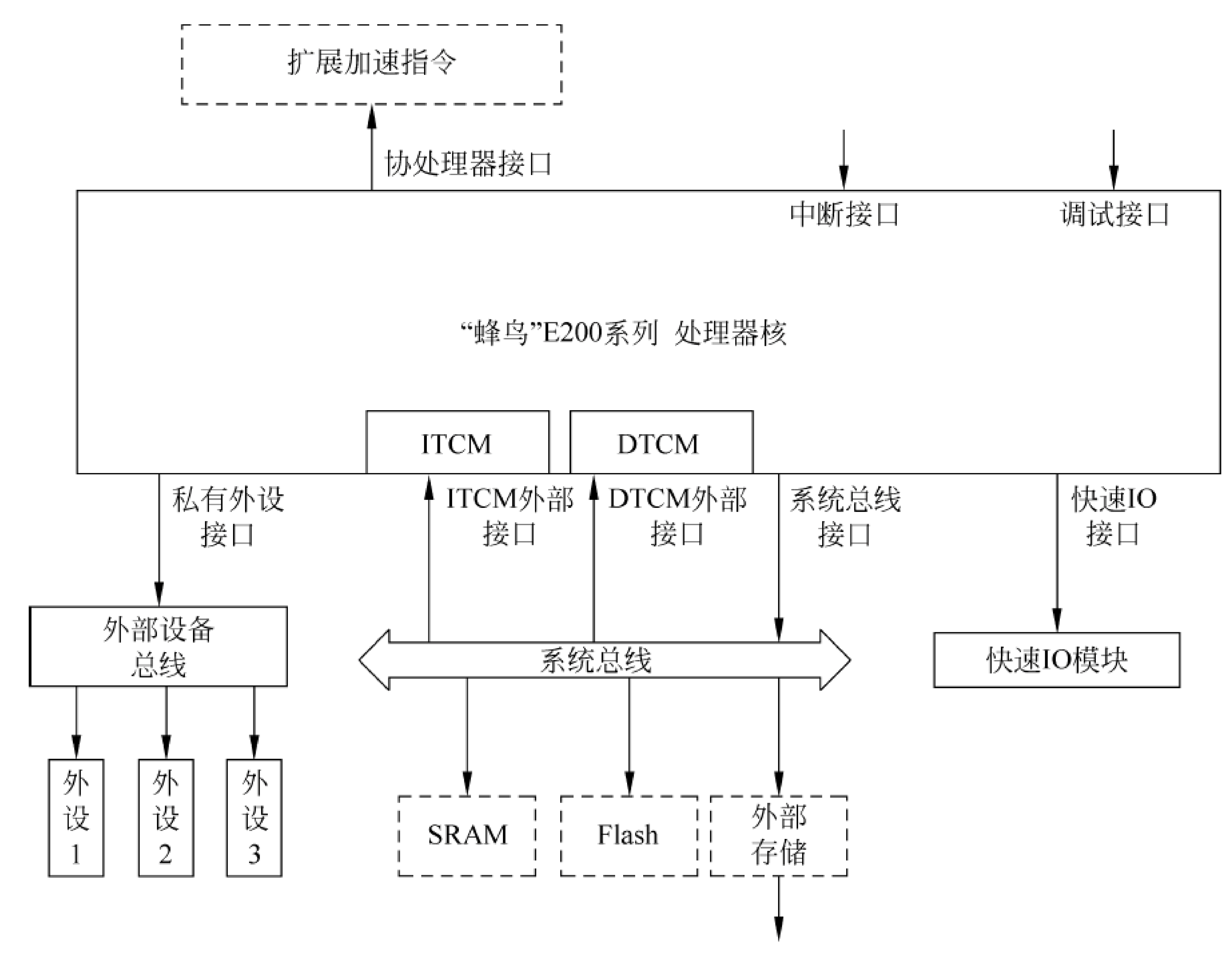


图 7-32 蜂鸟 E200 处理器系统示意图

表 7-8 处理器参数对比

项目	ARM Cortex-M0	ARM Cortex-M0+	ARM Cortex-M3	蜂鸟 E201	蜂鸟 E203	蜂鸟 E205
Dhrystone /(DMIPS/MHz)	0.84（官方） 1.21（经选项最大优化后的数据）	0.94（官方） 1.31（经选项最大优化后的数据）	1.25	1.171	1.262	1.355
CoreMark /(CoreMark/MHz)	2.33	2.42	3.32	1.352	2.226	3.327
最小配置逻辑门数/千逻辑门	12000	12000	36000	10000	12000	20000
频率	不详			180nm SLMC 工艺下 50～100MHz		
流水线深度	3	2	3	2	2	2
乘法器	有	有	有	无	有	有
除法器	无	无	有	无	有	有
ITCM(指令紧耦合存储) DTCM(数据紧耦合存储)	不提供专用 ITCM 与 DTCM 接口		提供 32 位的专用 ITCM 与 DTCM 接口	提供 64 位的专用 ITCM 接口 提供 32 位的专用 DTCM 接口		

续表

项　　目	ARM Cortex-M0	ARM Cortex-M0＋	ARM Cortex-M3	蜂鸟 E201	蜂鸟 E203	蜂鸟 E205
ECC 保护 SRAM	不提供 ECC 保护的 Memory			使用 ECC 保护 ITCM 和 DTCM		
可扩展性	不支持指令集扩展			可以进行指令集扩展（支持协处理器接口）		

注：

（1）Cortex-M0＋的乘法器可以配置成单周期乘法器或多周期迭代乘法器，因此其 Dhrystone 性能数据有两组；CoreMark 性能数据采用单周期乘法或多周期乘法器信息不详。

（2）本表格中有关 ARM-Cortex-M 系列处理器核的性能数据来自其他公开信息，非官方数据。请以 ARM 最新官方数据为准。

很多开源处理器核仅有内核的实现。在这种情况下为了使其能够被完整使用起来，用户需要花费不少精力来构建完整的 SoC 平台、FPGA 平台，并且调试的支持难度很大。蜂鸟 E200 不仅设计完成了内核的实现，还搭配完整的 SoC 平台，如图 7-33 所示。

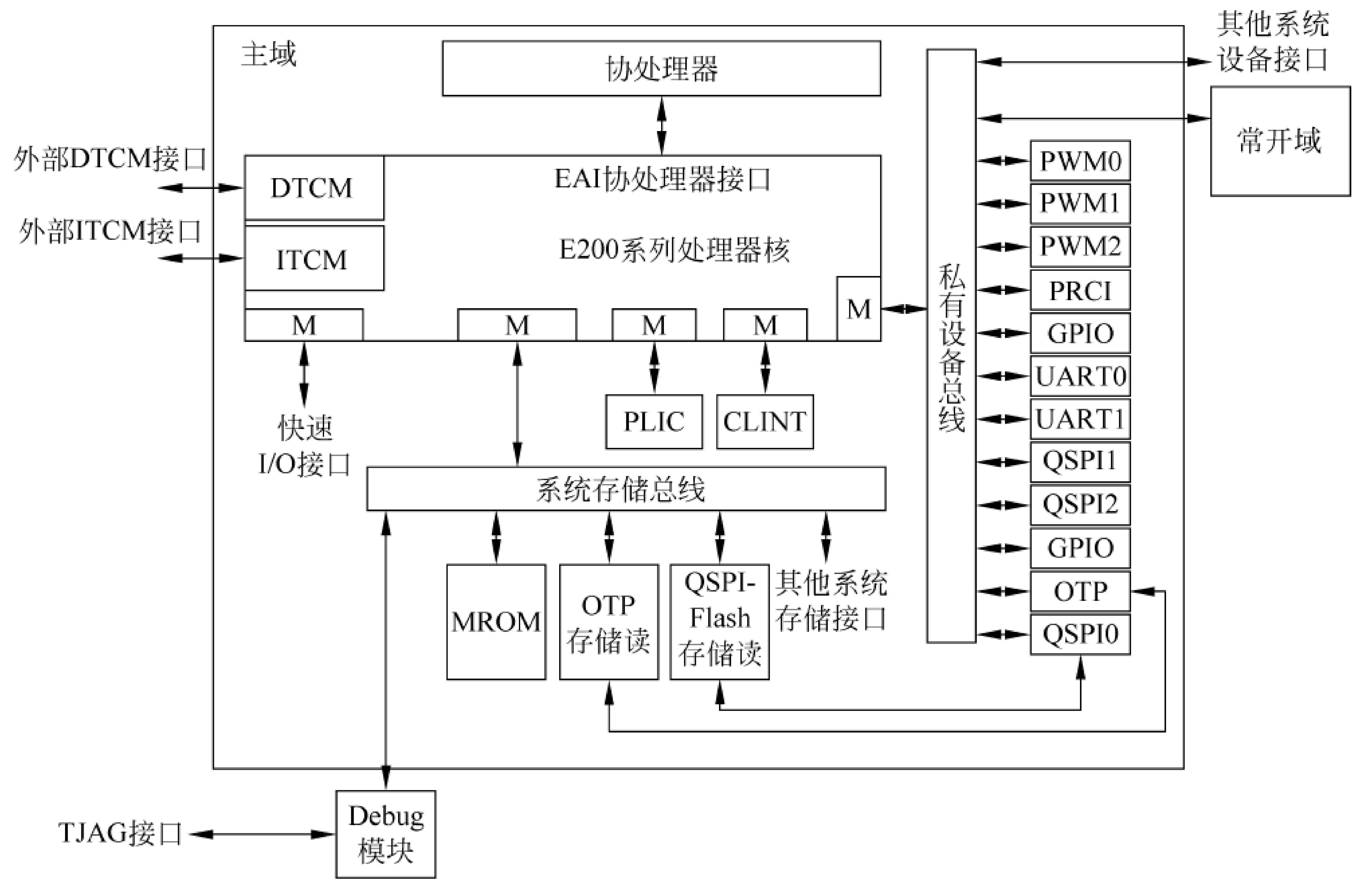


图 7-33　E200 系列处理器配套的 SoC

蜂鸟 E200 系列中，目前开源的具体型号为蜂鸟 E203 处理器核。蜂鸟 E200 系列中的每款处理器均有一定的可配置性。以开源的蜂鸟 E203 核为例，通过修改其目录下的 config. v 文件中的宏定义便可以实现不同的配置。config. v 文件在 e200_opensource 目录的结构如图 7-34 所示。

```
e200_opensource
 |----rtl                              // 存放 RTL 的目录
      |----e203                        // E203 核和 SoC 的 RTL 目录
           |----core                   // 存放 core 相关模块的 RTL 代码
                |----config.v          // 设定配置的源文件
```

图 7-34　config 文件目录

7.4 基于开源项目的 CPU 综合实验

如前所述，开源开放是信息技术发展的趋势。高校培养的学生也应该尽早地学习如何合理地利用开源项目完成开发需求，并基于开源项目做更多的创新工作，而不是每次都从零开始重复地造轮子。通过引入开源项目，还可以让学生有机会学习来自工业界工程师们的代码设计，学习其设计风格和思想。正如目前的操作系统实验，常采用把 Linux 或 μC/OS-Ⅲ等开源操作系统内核中某些功能和代码片段拿掉，让学生填补起来的方式，使学生掌握操作系统的设计原理，而不是都从零开始写一个自己的操作系统。在指令集开源的大背景下，组成原理的实验也完全可以借鉴这样的方法，在开源项目的基础上通过合理的设计，使学生达到学习目标。本书认为，基于蜂鸟 E200 RISC-V 开源处理器项目，至少可以设计出如下三种不同的实验方式。

7.4.1 从完整 SoC 项目中抽取出 CPU 内核上板验证

蜂鸟 E200 开源项目提供了完整的 SoC 实现，并可在 Arty 开发板上运行。但如果要在其他开发板上运行，还需要一定的移植工作。此外，如果对于组成原理所需单纯的 CPU 实验，则需要从完整 SoC 项目中提取出 CPU 内核。项目本身为内核部分提供了仿真验证，但如果要在本书所用的 PYNQ 板卡上或其他 FPGA 平台上进行内核的调试，还需要能够理解代码并对其进行相应的输入/输出工作。

因此，如果基于蜂鸟 E200 开源项目进行 CPU 实验，可以将开源网站及开源项目发布给学生，同时指定在 PYNQ 上对其提取出的内核进行验证的接口和指令。学生可以在理解开源项目的基础上完成内核的移植和验证。

该实验要求学生理解开源项目的代码并对其进行修改和验证，具备一定的难度，适合作为集中性实践环节的实验任务。

7.4.2 删减掉特定的部分并补全

第二种实验方式可以通过把完整的 CPU 内核的特定部分删减掉并让学生对其进行补全。该实验提供一个不完整的蜂鸟 E203 处理器项目，缺少的部分都位于 E203 处理器的流水线第 2 级——执行阶段，该阶段缺少部分指令的译码和执行。学生需要将蜂鸟 E203 的内核中缺少的部分补充完整，使 E203 处理器能正常运行，通过仿真方式对处理器进行功能验证。

1. 仿真测试要求

在测试程序中写入任意 5 个数，找出大于 10 的数并将其个数写回寄存器 1 中，找出小于 10 的数并其个数写回寄存器 2 中，通过仿真结果观察处理器能否正常工作。

2. 仿真测试流程

首先将处理器补充完整，可以先以汇编指令形式编写指令测试代码，再将汇编指令转换成 RISC-V 指令形式。将编写好的指令逐条写入到处理器的指令存储器中。对项目编写 Testbench 文件添加时钟激励以运行处理器，观察测试结果是否正确。

7.4.3　扩展开源处理器的流水线级数

开源的蜂鸟 E203 处理器为两级流水线结构，因此可以要求学生在理解开源代码及流水线原理的基础上对蜂鸟 E203 处理器的流水线进行扩展，如将两级流水线扩展为 3 级流水线。观察修改后的仿真波形与 2 级流水线进行对比，观察其结果是否一致，运算速度是否发生变化。该实验也具备一定的难度，适合作为集中性实践环节的实验任务。

7.4.4　优秀工作的遴选方法

对于综合性的组成原理实验，由于通常具有一定的难度，学生很难在规定的时间内在课内完成并进行演示，大量的工作需要在课外完成。因此使用线上平台在课外进行实验就显得比较重要。但是，学生在课外进行实验会带来评价上的难度。教师无法判断学生提交的工作是自己独立完成、与其他同学讨论后完成，还是单纯地借鉴了同学的工作。即使通过源代码的查看也难以判断，因为本身就是基于开源项目做的。

因此本书建议可通过几个阶段和方法进行评价。当然，由于水平和思路所限，不见得是最佳的方式，仅希望起到抛砖引玉的作用。

1. 通过性测试

通过性测试即验证学生提交的实验，功能是否正确，完成程度怎么样。测试有几种不同的方式：对于学生提交的实验结果，教师可通过提供标准的 Testbench 对其进行仿真验证，但这是一个极其耗时的过程。目前同济大学的组成原理相关实验采用用了这种方式，通过在后台服务器运行仿真程序完成。此外，对于基于本书所介绍的 PYNQ 平台的实验，教师完全可编写 Python 代码对学生的比特流进行测试，这样的测试是针对真实运行的系统，更具备准确性。当然，无论采用哪种方式，对学生的实验进行一一测试，都是耗时耗力的工作。最好的方式是有自动评测的方法，通过对学生提交作业的自动的评测，可以将教师从琐碎的重复性工作中解放出来。自动评测的功能可能需要在线上平台支持下才能实现，因为其可以提供管理功能。有兴趣的读者可以关注这方面的发展。

2. 实验报告评阅

在提交的实验通过性检测的基础上，已经可以对学生的完成程度有基本的了解。在此基础上可以进一步阅读学生的实验报告，主要是了解学生对所做实验的理解和认识。因为虽然完成同样的实验，但不同的同学通过实践之后对理论知识的理解和对整个事情的认识差异程度还是很大的。这个可以通过实验报告的评阅得以了解。

3. 优秀作品答辩与展示

经过前面两步，已经可以对学生的完成程度有较好的认识。但如果要进一步从中选出优秀的作品，可以采用答辩的方式。如果全员答辩有困难，可以鼓励同学们进行答辩申请，只有参加答辩的同学才有可能获得优秀。通常来说，确实只有实验完成好、理解透彻的同学才会申请答辩。以此可以区分是独立完成还是在他人的帮助下才完成的。同时，学生的表达和沟通能力也是工程教育中对培养学生的要求。

第 8 章 实验内容设计

CHAPTER 8

本章给出了基于 PYNQ 的计算机组成原理实验的参考内容安排。考虑到不同学校的实际情况，为便于直接选取到合适的实验内容，本书给出了基于原理图和基于 HDL 的两种方式，同时提供了从基本的数字逻辑到综合性的组成原理实验的具有一定跨度的参考实验内容。当然，本书给出的实验内容仅作为参考，读者完全可在前面介绍的工具和方法的基础上，灵活安排适合自己的实验内容。

为了方便进行实验，书中提供了一个连接完成的 AxiGPIO 交互模块，用户只需要通过 Vivado 的 tcl 指令调用该 AxiGPIO 交互模块的 tcl 文件就可以直接使用。具体资料请见源代码的对应目录。AxiGPIO 交互模块包含 ZYNQ 芯片 IP、系统复位 IP、Axi 互联 IP 及 10 个 AxiGPIO 模块这几部分，其原理如图 8-1 所示。

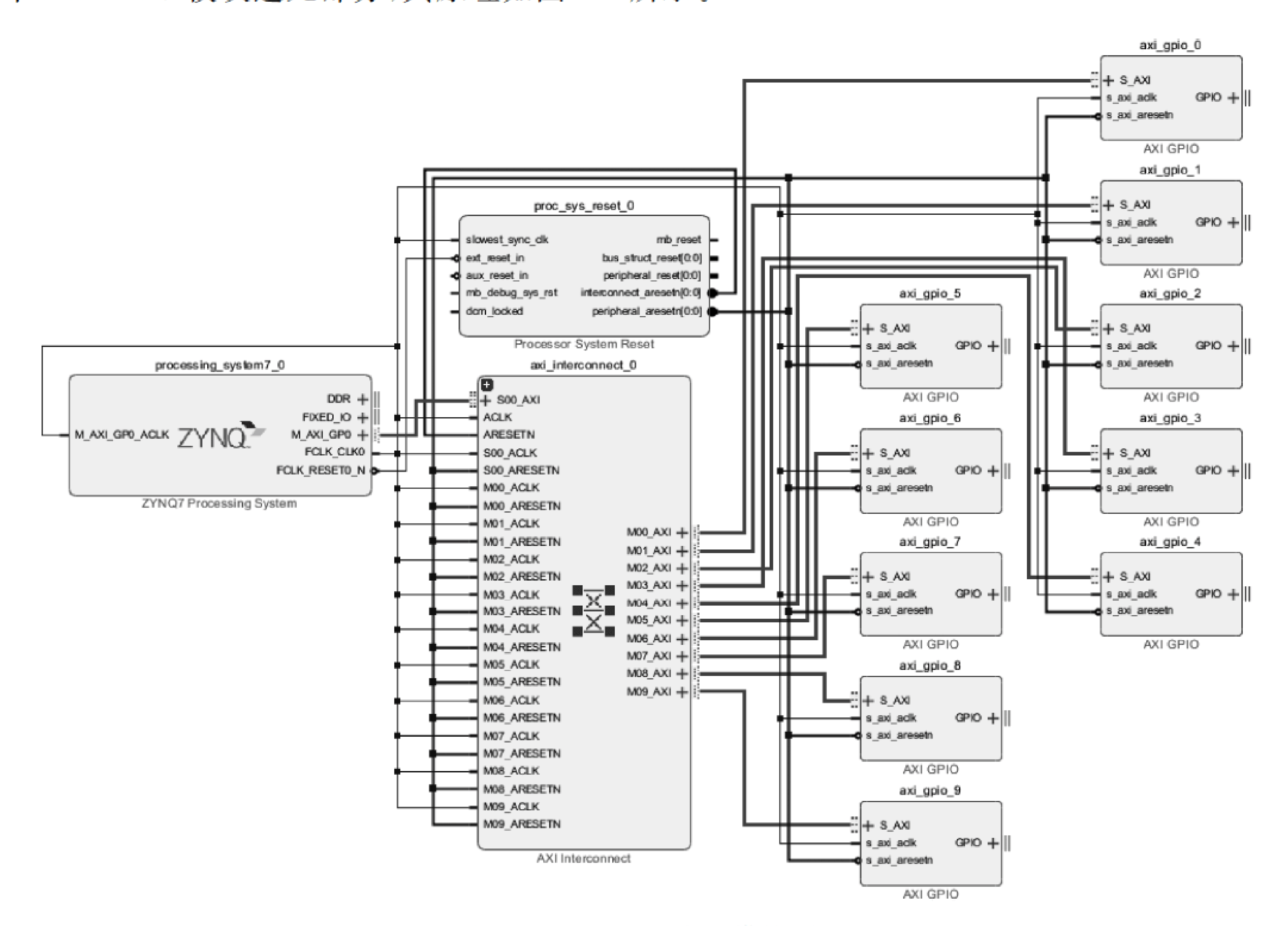


图 8-1 AxiGPIO 交互模块原理

AxiGPIO 模块建立了 ARM 芯片与 AxiGPIO 的连接，用户可以参考第 4 章 AxiGPIO 使用部分的内容，将自己设计的模块引脚与 AxiGPIO 模块进行连接，以便和运行在 PS 端的 Python 代码交互，从而完成整个工程的设计。

8.1 基于原理图的实验

采用原理图的方式进行组成原理的实验，其优点是比较直观，易于理解硬件的连接方式，方便形成硬件思维，而且不需要进行专门的学习。原理图的方式比较适用于初学者，可帮助他们更好地理解硬件。同时对数字逻辑及组成原理部件实验等复杂程度不是很高的系统，设计效率也比较高。对于之前没有 FPGA 开发基础的学校来说，先以原理图方式开始，不失为一种比较平缓的切入方式。

但原理图方式的缺陷是：原理图仅描述的是硬件的结构，而对于构成的硬件的行为并不进行直接的展示。另外，对于复杂程度较高的综合性组成原理实验，如完整的 CPU 的设计，采用原理图开发时效率就会大幅下降。而硬件描述语言 Verilog/VHDL 与原理图恰好相反，其对于初学者来说具有较高的门槛，需要有一定的学习成本。并且对于没有硬件思维的用户，容易将其与软件编程混为一谈，描述出不能综合的电路系统。但其优点是：适合于描述复杂的硬件系统，对于熟练掌握者，通过代码复用等方法也可以具有很高的开发效率。同时，HDL 可以进行硬件的行为描述，方便读者理解硬件的功能。

因此，本节所设计的基于原理图的实验具体开展方式如下：

实验指导书中以 Verilog HDL 源代码(.v 文件)的方式为学生提供每个实验所需的基础模块。学生在开展实验时需要根据第 4 章所介绍的方法自行把 HDL 源文件生成原理图的方式供实验用。使用这些基础模块，再用原理图的方式完成整个实验。

以此方式，学生可以以原理图的方式完成整体硬件系统的搭建，同时可以通过参考 HDL 源代码的方式更好地理解每个基础模块的行为和功能。

8.1.1 全加器

1. 实验目的

掌握基于 PYNQ 平台及 Vivado 工具的组合逻辑电路的开发流程及调试方式。了解 1 位全加器的基本构成和工作原理，能够在 1 位全加器基础上搭建多位全加器。

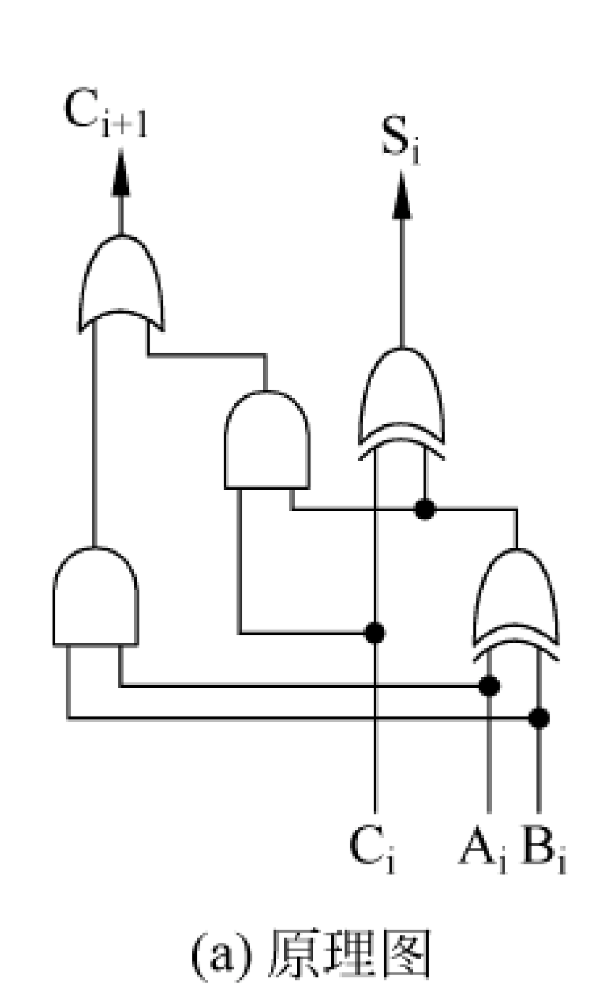
2. 实验要求

要求采用源代码中对应目录下的“基础模块”所提供的“.v”源文件生成 IP，并采用原理图的方式调用所生成的 IP 设计所要求的系统。采用 Vivado 进行仿真，并进行 I/O 的连接，最后生成流文件，并下板完成验证。

3. 实验说明

1) 实验原理

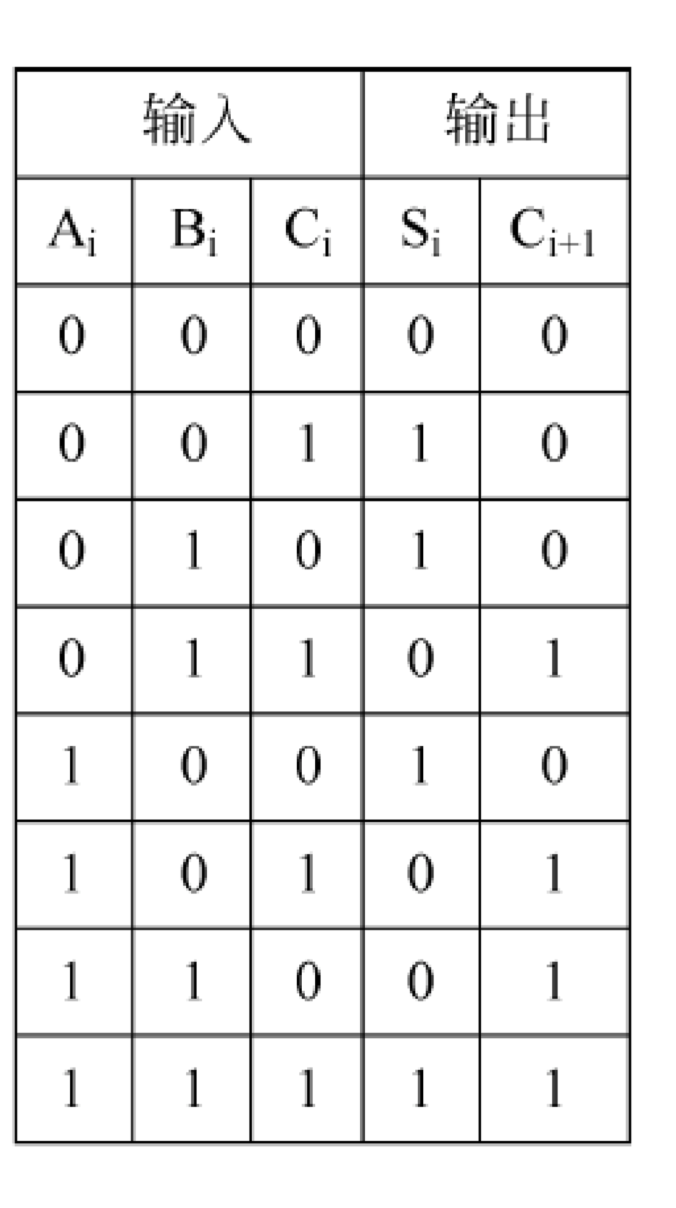
一位全加器的原理图、逻辑表达式及对应的真值表如图 8-2 所示。可以先完成一位全加器的实验，再视情况完成 n 位全加器的设计实验，如图 8-3 所示。其中 FA 代表图 8-2 中的一位全加器。



(a) 原理图

$S_i=A_i \oplus B_i \oplus C_i$
$C_{i+1}=A_iB_i+A_iC_i+B_iC_i$
$=A_iB_i+(A_i \oplus B_i)C_i$

(b) 逻辑表达式



输入			输出	
A_i	B_i	C_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(c) 真值表

图 8-2　一位全加器

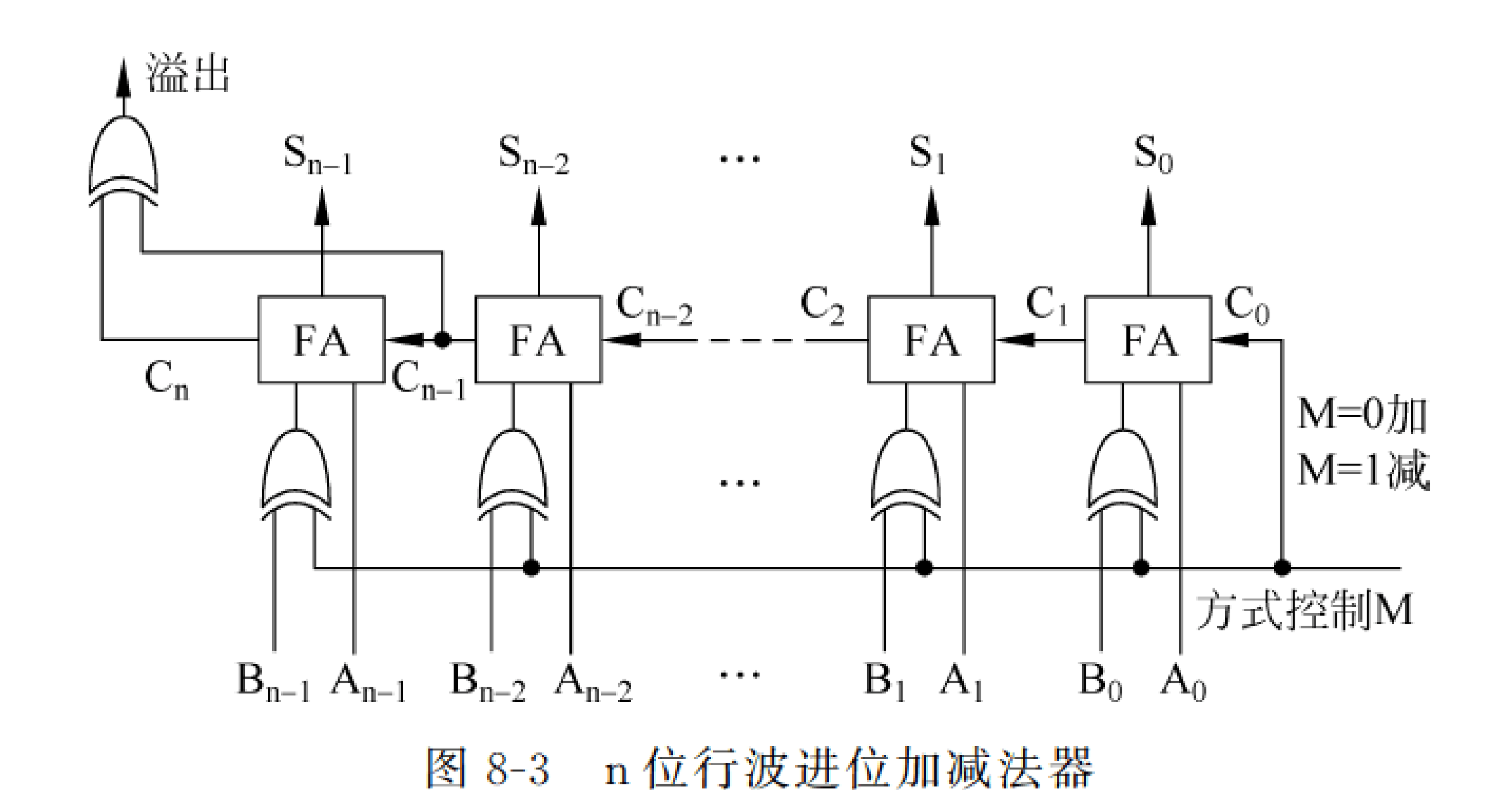


图 8-3　n 位行波进位加减法器

2）Python 交互模板

（1）加载比特流（一位加法器的 Python 交互模板）的代码：

```
from pynq import Overlay
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/....bit")#bit文件路径
```

（2）GPIO 口的命名要与原理图的连接一致。代码如下：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_Ci = AxiGPIO(gpio_0).channel1
gpio_Ai = AxiGPIO(gpio_0).channel2
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_Bi = AxiGPIO(gpio_1).channel1
gpio_5 = overlay.ip_dict['axi_gpio_5']
gpio_Ci1 = AxiGPIO(gpio_5).channel1
gpio_Si = AxiGPIO(gpio_5).channel2
```

（3）加法器功能验证（GPIO 口的读写）的代码：

```
# 写GPIO需要和其位数相匹配的掩码，如gpio_Ci是1位的，那么它的掩码2进制数0b1
mask  = 0b1
gpio_Ci.write(0b1,mask)
gpio_Ai.write(0b1,mask)
```

```
gpio_Bi.write(0b1,mask)

Ci1=gpio_Ci1.read()
Si=gpio_Si.read()
print(bin(Ci1))
print(bin(Si))
```

3）源代码

本书配套源代码中，为实验提供了如下材料：

(1)“基础模块”文件夹：该文件夹下提供的是每个实验所需的基础模块.v 源文件，在采用原理图方式进行实验时，用户可以将其生成 IP 备用。

(2)“test.v”：是对用户完成的设计进行仿真时所需的 Testbench 示例程序，用户可直接使用，也可以根据需要自行修改。

(3)“xxx.ipynb”是本节“Python 交互模板”部分所介绍的 Python 源文件。

4. 实验步骤

(1) 将本实验对应源代码中提供的基本模块的.v 文件生成原理图。

(2) 将基础模块进行连接，并使用本实验对应源代码中的仿真文件进行仿真。

(3) I/O 处理，将系统的逻辑 I/O 与 PS 相连接，以便进行调试。

(4) 重新综合、实现及生成比特流文件，并将比特流文件及对应的 tcl 文件上传到所用 PYNQ 节点。

(5) 下板实验，使用本实验对应源代码中的 Python 调试文件，观察真实运行后的系统是否正常工作。

8.1.2 译码器

1. 实验目的

掌握基于 PYNQ 平台及 Vivado 工具的组合逻辑电路的开发流程及调试方式。了解二—四译码器的基本构成和工作原理，并在此基础上实现三—八译码器。

2. 实验要求

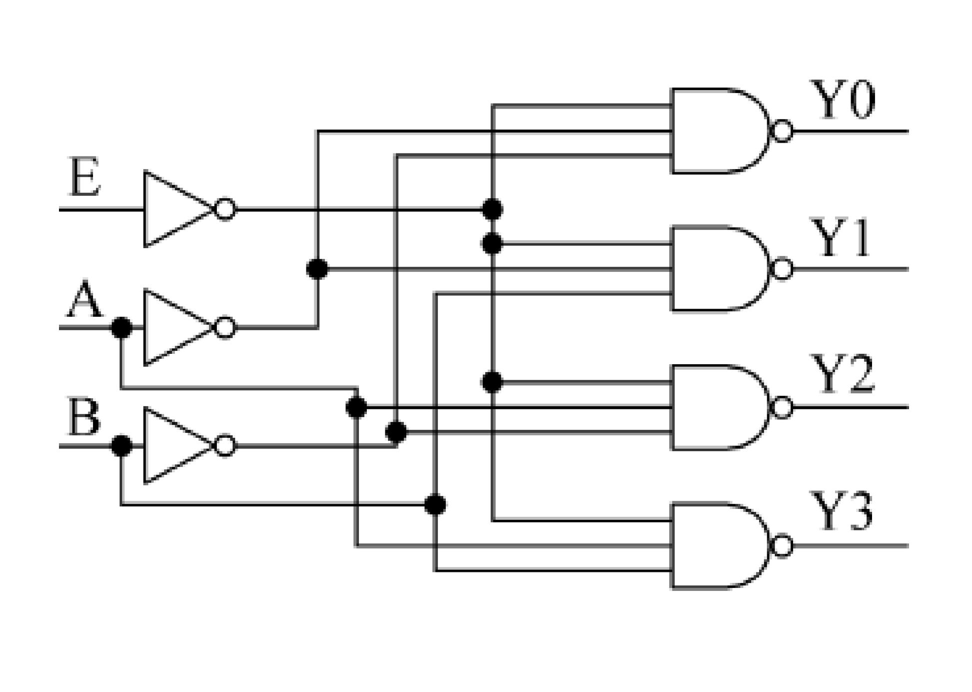
要求采用源代码中对应目录下的“基础模块”所提供的“.v”源文件生成 IP，并采用原理图的方式调用所生成的 IP 设计所要求的系统。采用 Vivado 进行仿真，并进行 I/O 的连接，最后生成比特流文件，并下板完成验证。

3. 实验说明

1）实验原理

二—四译码器的原理图及对应的真值表如图 8-4 所示。Y3、Y2、Y1 和 Y0 为译码器输出端。E 为使能端，低电平有效。当 E=0 时，允许译码器工作，Y3、Y2、Y1、Y0 中仅有一个为低电平输出。

基于二—四译码器的基本构成和工作原理，完成三—八译码器实验。三—八译码器的真值表如图 8-5 所示。



(a) 原理图

输入			输出			
E	A	B	Y0	Y1	Y2	Y3
1	×	×	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

(b) 真值表

图 8-4　二—四译码器

输入				输出							
E	A	B	C	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
1	×	×	×	1	1	1	1	1	1	1	1
0	0	0	0	0	1	1	1	1	1	1	1
0	0	0	1	1	0	1	1	1	1	1	1
0	0	1	0	1	1	0	1	1	1	1	1
0	0	1	1	1	1	1	0	1	1	1	1
0	1	0	0	1	1	1	1	0	1	1	1
0	1	0	1	1	1	1	1	1	0	1	1
0	1	1	0	1	1	1	1	1	1	0	1
0	1	1	1	1	1	1	1	1	1	1	0

图 8-5　三—八译码器真值表

2）Python 交互模板

（1）加载比特流（二—四译码器的 Python 交互模板）的代码：

```
from pynq import Overlay
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/....bit")
```

（2）GPIO 口的命名要与原理图的连接一致，代码如下：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_E = AxiGPIO(gpio_0).channel1
gpio_A = AxiGPIO(gpio_0).channel2
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_B = AxiGPIO(gpio_1).channel1
gpio_5 = overlay.ip_dict['axi_gpio_5']
gpio_Y0 = AxiGPIO(gpio_5).channel1
gpio_Y1 = AxiGPIO(gpio_5).channel2
gpio_6 = overlay.ip_dict['axi_gpio_6']
gpio_Y2 = AxiGPIO(gpio_6).channel1
gpio_Y3 = AxiGPIO(gpio_6).channel2
```

（3）二—四译码器的功能验证代码：

```
mask = 0xb1
gpio_E.write(0,mask)
gpio_A.write(0,mask)
gpio_B.write(0,mask)

Y0=gpio_Y0.read()
Y1=gpio_Y1.read()
Y2=gpio_Y2.read()
Y3=gpio_Y3.read()
print(Y0,Y1,Y2,Y3)
```

3）源代码

本实验源代码中所含文件介绍同 8.1.1 节。

4. 实验步骤

实验步骤同 8.1.1 节。

8.1.3　多路选择器

1. 实验目的

掌握基于 PYNQ 平台及 Vivado 工具的组合逻辑电路的开发流程及调试方式。了解一位二选一多路选择器的基本构成和工作原理，能够在此二选一多路选择器的基础上搭建一位四选一多路选择器。

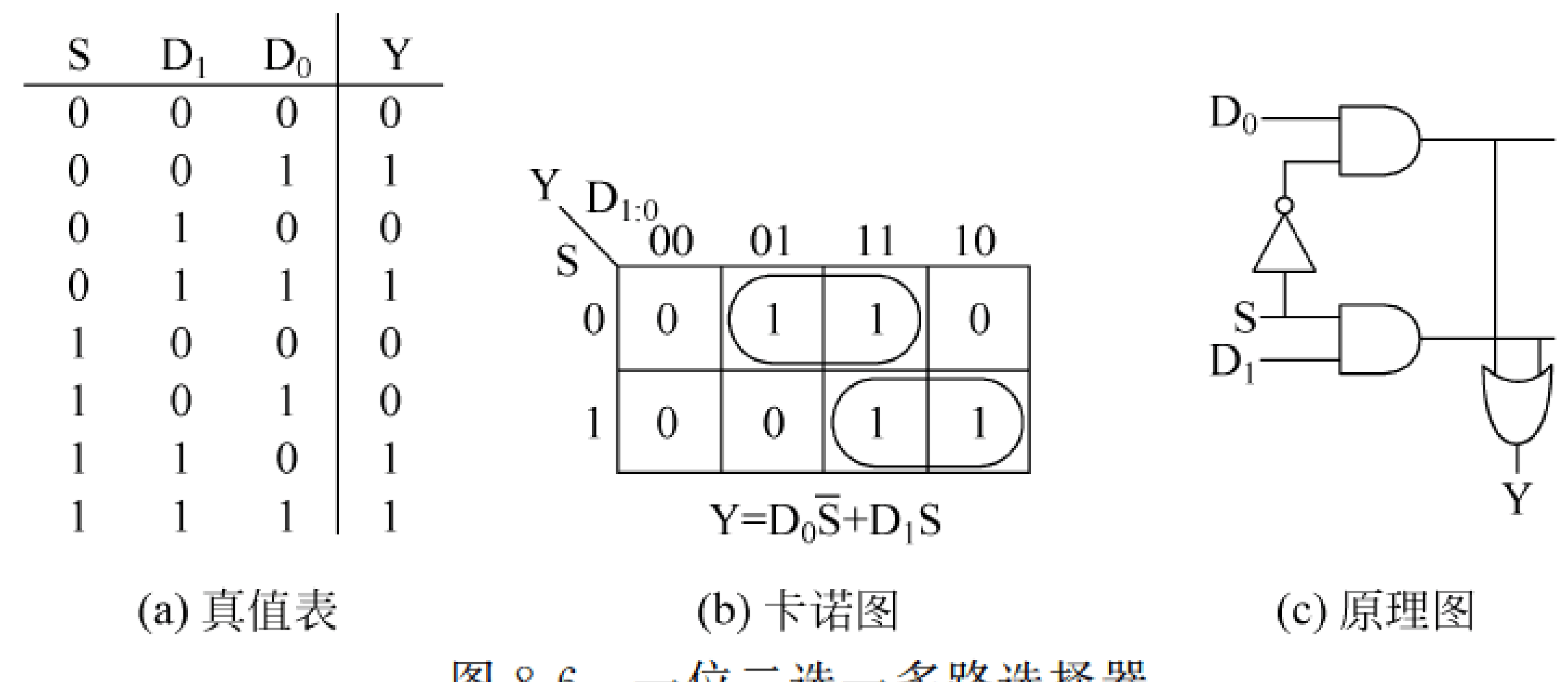
2. 实验要求

要求采用源代码中对应目录下的“基础模块”所提供的“.v”源文件生成 IP，并采用原理图的方式调用所生成的 IP 设计所要求的系统。采用 Vivado 进行仿真，并进行 I/O 的连接，最后生成比特流文件，并下板完成验证。

3. 实验说明

1）实验原理

一位二选一多路选择器的真值表、卡诺图及对应的原理图如图 8-6 所示。



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

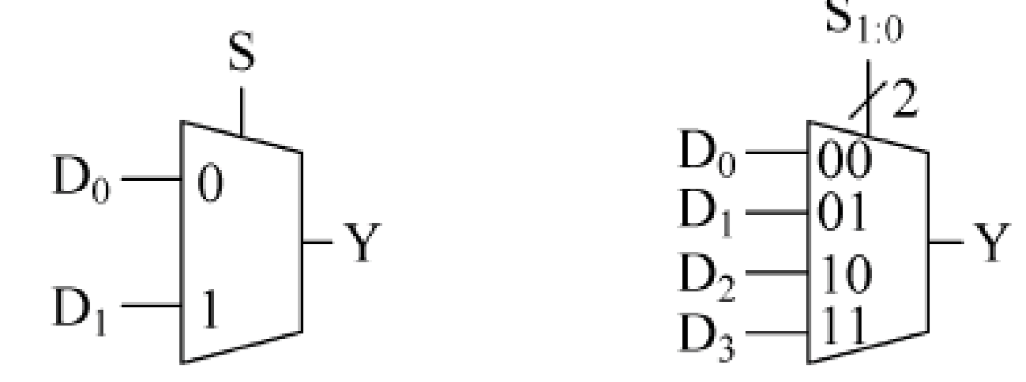
(a) 真值表　(b) 卡诺图　(c) 原理图

图 8-6　一位二选一多路选择器

它有 2 个输入 D_0 和 D_1，一个选择输入 S 和一个输出 Y。二选一多路选择器根据选择信号的值在 2 个输入数据中选择一个数据输出。一个二选一多路选择器可以用与或两级逻辑实现，其布尔表达式可以通过卡诺图或者分析得到。

先完成一位二选一多路选择器的实验，再视情况完成一位四选一多路选择器的设计实验（一个 N 选一多路选择器需要 $\log_2 N$ 条选择线）。四选一多路选择器可以使用与或两级逻辑实现，也可以使用多个二选一多路选择器实现。

一位二选一多路选择器和一位四选一多路选择器的电路符号如图 8-7 所示。



(a) 二选一多路选择器　(b) 四选一多路选择器

图 8-7　多路选择器

2）Python 交互模板

（1）加载比特流（二选一多路选择器的 Python 交互模板）的程序代码：

```
from pynq import Overlay
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/wzg/....bit")
```

（2）GPIO 口的命名要与原理图的连接一致，代码如下：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_D0 = AxiGPIO(gpio_0).channel1
gpio_S = AxiGPIO(gpio_0).channel2
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_D1 = AxiGPIO(gpio_1).channel1
gpio_5 = overlay.ip_dict['axi_gpio_5']
gpio_Y = AxiGPIO(gpio_5).channel1
```

（3）二选一多路选择器的功能验证的程序代码：

```
mask  = 0b1
gpio_D0.write(0,mask)
gpio_D1.write(1,mask)
gpio_S.write(1,mask)

Y=gpio_Y.read()
print(Y)
```

3）源代码

本实验源代码中所含文件介绍同 8.1.1 节。

4. 实验步骤

实验步骤同 8.1.1 节。

8.1.4 触发器与寄存器

1. 实验目的

掌握基于 PYNQ 平台及 Vivado 工具的时序逻辑电路的开发流程及调试方式。了解 SR 锁存器、D 锁存器和 D 触发器的基本构成和工作原理，能够在 D 触发器的基础上进一步设计带使能端和同步复位功能的触发器，并最终完成搭建一个带使能端和同步复位功能的 2 位寄存器的实验。

2. 实验要求

要求采用源代码中对应目录下的“基础模块”所提供的“.v”源文件生成 IP，并采用原理图的方式调用所生成的 IP 设计所要求的系统。采用 Vivado 进行仿真，并进行 I/O 的连接，最后生成流文件，并下板完成验证。

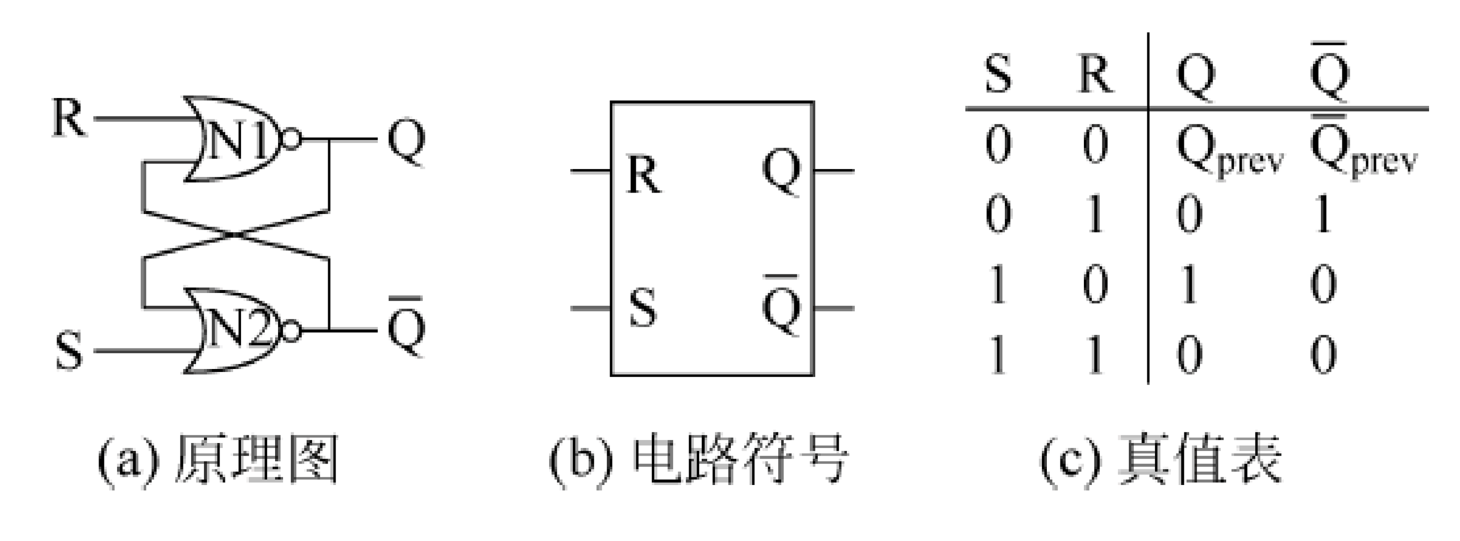
3. 实验说明

1）实验原理

（1）SR 锁存器：SR 锁存器由一对交叉耦合的或非门组成，其原理图、电路符号以及真值表如图 8-8 所示。SR 锁存器有两个输入 S 和 R，两个输出 Q 和 $\overline{Q}$。输入 S 和 R 分别表示置位和复位。置位表示将 Q 设置为 1，复位表示将 Q 设置为 0。当输入 S 和 R 均为无效时，Q 将保持原来的值 Q_{pre} 不变。S 和 R 同时有效是没有意义的，这样会产生两个输出为 0 的混乱电路响应。

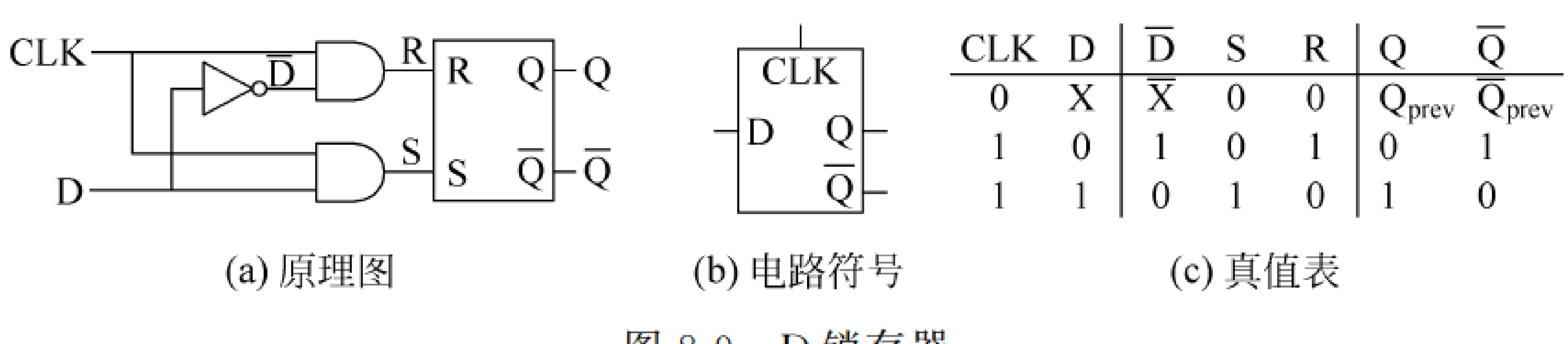
（2）D 锁存器：D 锁存器的原理图、电路符号以及真值表如图 8-9 所示。D 锁存器避免了 S 和 R 同时有效的情况。当 CLK=0 时，D 锁存器是不透明的，无论 D 为何值，有 S=R=0，输出 Q 保持原来的值 Q_{pre}。当 CLK=1 时，D 锁存器是透明的，数据 D 通过 D 锁存器流向 Q，有 Q=D。

（3）D 触发器：一个 D 触发器可以由反相时钟控制的两个背靠背的 D 锁存器构成，它



(a) 原理图　(b) 电路符号　(c) 真值表

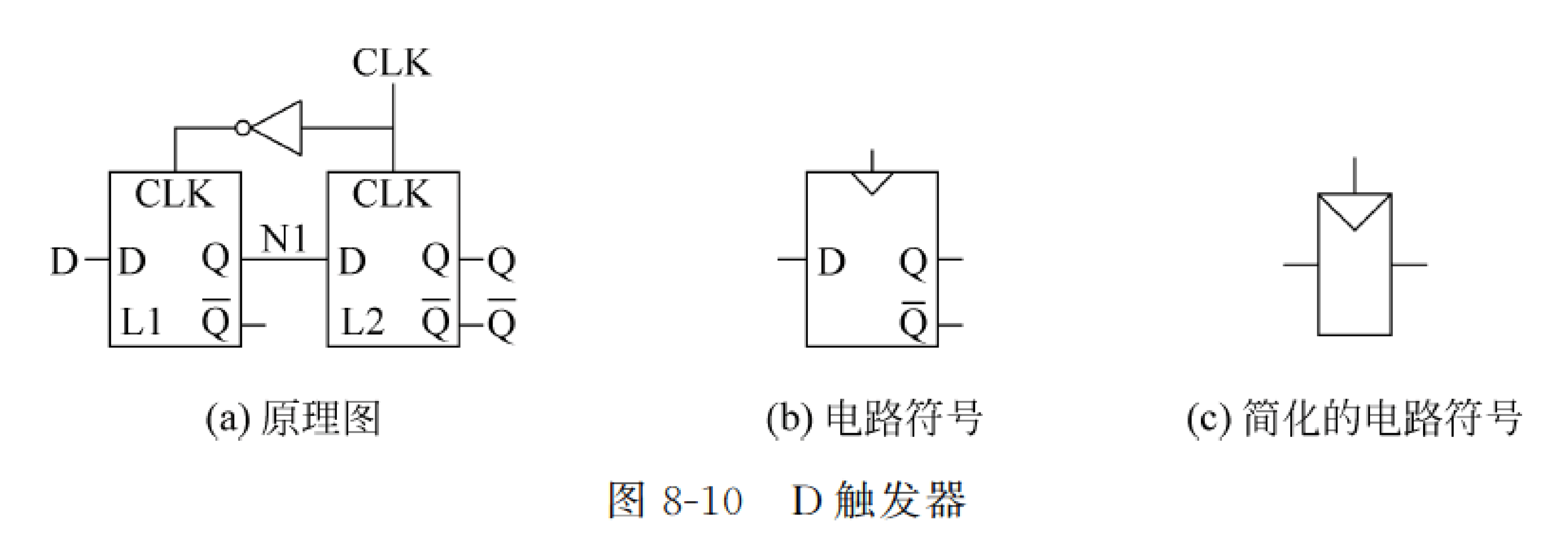
图 8-8　SR 锁存器



(a) 原理图　(b) 电路符号　(c) 真值表

图 8-9　D 锁存器

的原理图、电路符号及简化的电路符号如图 8-10 所示。D 触发器在时钟上升沿将 D 复制到 Q，在其他时间 D 触发器保持原来的状态。当 CLK＝0 时，锁存器 L1 是透明的，锁存器 L2 是不透明的，D 的值传递到 N1；当 CLK＝1 时，锁存器 L1 是不透明的，锁存器 L2 是透明的，N1 的值传递到 Q；在其他时刻，因为总有一个阻塞 D 到 Q 的锁存器，所以 Q 保持原来的值。



(a) 原理图　(b) 电路符号　(c) 简化的电路符号

图 8-10　D 触发器

（4）带使能端的触发器：带使能端的触发器增加一个称为 EN 的输入，该输入用于确定时钟沿是否载入数据。当 EN＝1 时，带使能端的触发器与普通 D 触发器一样；当 EN＝0 时，带使能端的触发器忽略时钟，保持原来的状态、当希望在某些时钟沿而不是所有时钟沿载入一个新值时，带使能端的触发器就派上了用场。

注意：不要在时钟上执行逻辑（在时钟上改动），因为可能会使时钟延时并导致时序错误。

（5）带同步复位功能的触发器：带同步复位功能的触发器增加了一个称为 RESET 的输入。RESET 是一个低电平有效的信号，即当其为 0 时执行复位功能。当 RESET＝0 时，带同步复位功能的触发器忽略输入 D 并在时钟上升沿将输出 Q 复位为 0；当 RESET＝1 时，带同步复位功能的触发器与普通 D 触发器一样。

（6）带使能端和同步复位功能的触发器。

带使能端和同步复位功能的触发器增加了两个输入 EN 和 RESET。EN 的优先级比 RESET 高，即 EN 为 1 时，RESET 无效。

2）Python 交互模板

（1）加载比特流（2 位寄存器的 Python 交互模板）的程序代码：

```
from pynq import Overlay
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/....bit")
```

(2) GPIO 的命名要与原理图的连接一致,其程序代码如下:

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_CLK = AxiGPIO(gpio_0).channel1
gpio_EN = AxiGPIO(gpio_0).channel2
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_RESET = AxiGPIO(gpio_1).channel1
gpio_D = AxiGPIO(gpio_1).channel2
gpio_5 = overlay.ip_dict['axi_gpio_5']
gpio_Q = AxiGPIO(gpio_5).channel1
```

(3) 验证寄存器功能的程序代码:

```
mask1  = 0b1
mask2  = 0b11
# 第一个时钟上升沿, EN=1, RESET=1, D=2'b10,验证Q是否为2'b10
gpio_CLK.write(0b0,mask1)
gpio_EN.write(0b1,mask1)
gpio_RESET.write(0b1,mask1)
gpio_D.write(0b10,mask2)
gpio_CLK.write(0b1,mask1)
Q=gpio_Q.read()
print (bin(Q))
```

```
# 第二个时钟上升沿, EN=0, RESET=0, D=2'b10,验证Q是否为2'b10
gpio_CLK.write(0b0,mask1)
gpio_EN.write(0b0,mask1)
gpio_RESET.write(0b0,mask1)
gpio_D.write(0b10,mask2)
gpio_CLK.write(0b1,mask1)
Q=gpio_Q.read()
print (bin(Q))
```

3) 源代码

本实验源代码中所含文件介绍同 8.1.1 节。

4. 实验步骤

实验步骤同 8.1.1 节。

8.1.5 移位寄存器

1. 实验目的

掌握基于 PYNQ 平台及 Vivado 工具的时序逻辑电路的开发流程及调试方式。了解移位寄存器的基本构成和工作原理,能够在此基础上将其改进成 3 位带并行加载的移位寄存器。

2. 实验要求

要求采用源代码中对应目录下的"基础模块"所提供的".v"源文件生成 IP,并采用原理图的方式调用所生成的 IP 设计所要求的系统。采用 Vivado 进行仿真,并进行 I/O 的连接,最后生成流文件,并下板完成验证。

3. 实验说明

1) 实验原理

(1) 移位寄存器:移位寄存器的原理图如图 8-11 所示,它由 N 个触发器串联而成。移位寄存器包括时钟、串行输入 S_{in}、串行输出 S_{out} 和 N 位并行输出 $Q_{N-1:0}$。在时钟的每一个上升沿,从 S_{in} 移入一个新的位,所有后续内容都向前移动。移位寄存器可以完成串行到并行的转换。输入由 S_{in} 以串行方式提供,在 N 个周期后,前面的 N 位输入可以在 Q 中并行访问。

(2) 带并行加载的移位寄存器:带并行加载的移位寄存器增加了并行输入 $D_{N-1:0}$ 和控

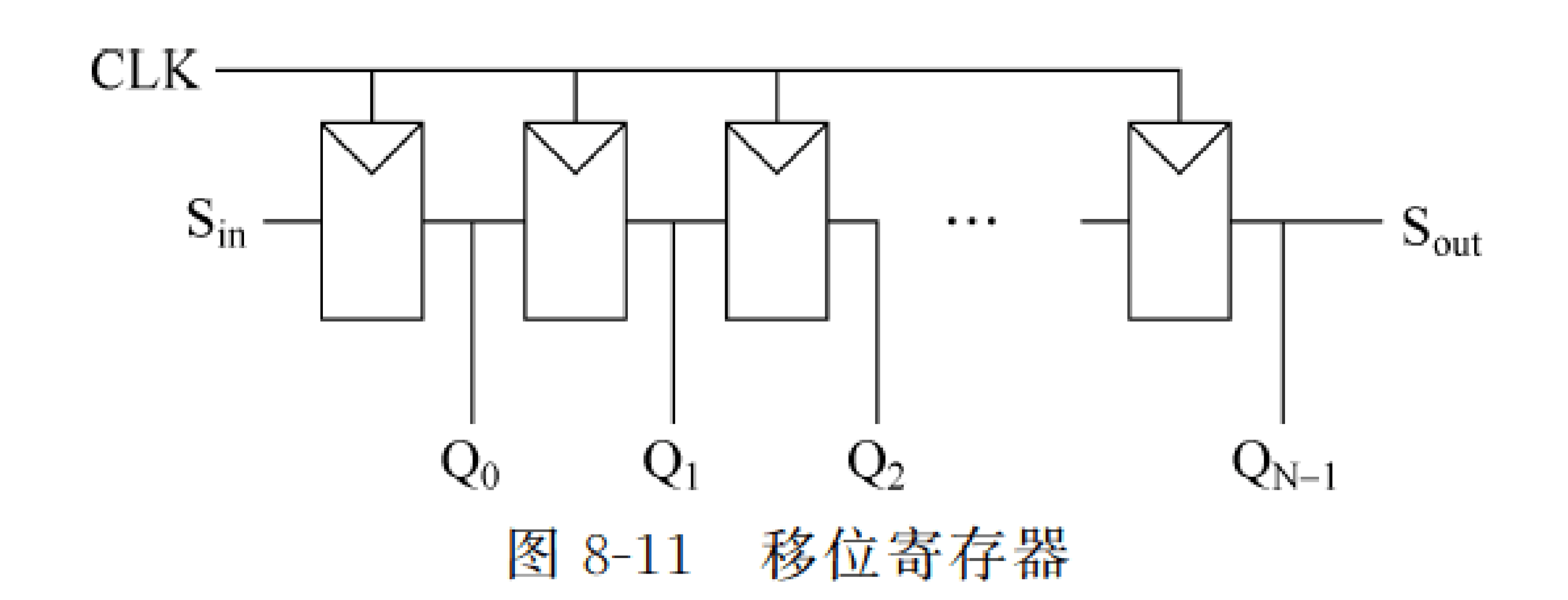


图 8-11　移位寄存器

制信号 Load，它既可以完成串行到并行的转换，也可以完成并行到串行的转换。当 Load 有效时，触发器从输入 D 中并行加载数据，否则移位寄存器正常移位，在 N 个周期后就将 $D_{N-1:0}$ 以串行的方式输出。

2）Python 交互模板

（1）加载比特流（带并行加载的移位寄存器的 Python 交互模板），代码如下：

```
from pynq import Overlay
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/....bit")
```

（2）GPIO 的命名要与原理图的连接一致，代码如下：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_Load = AxiGPIO(gpio_0).channel1
gpio_CLK = AxiGPIO(gpio_0).channel2
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_Sin = AxiGPIO(gpio_1).channel1
gpio_D = AxiGPIO(gpio_1).channel2
gpio_5 = overlay.ip_dict['axi_gpio_5']
gpio_Sout = AxiGPIO(gpio_5).channel1
gpio_Q = AxiGPIO(gpio_5).channel2
```

（3）验证移位寄存器的并行到串行的转换，代码如下：

```
# 写GPIO需要和其位数相匹配的掩码，如gpio_D是3位的，那么它的掩码二进制数0b11
mask  = 0b1
mask1  = 0b111
# 第一个周期加载并行数据D，输出Sout
gpio_CLK.write(0b0,mask)
gpio_Load.write(0b1,mask)
gpio_Sin.write(0b0,mask)
gpio_D.write(0b110,mask1)
gpio_CLK.write(0b1,mask)
Sout=gpio_Sout.read()
print(bin(Sout))

# 第二个周期，输出Sout
gpio_CLK.write(0b0,mask)
gpio_Load.write(0b0,mask)
gpio_Sin.write(0b0,mask)
gpio_D.write(0b000,mask1)
gpio_CLK.write(0b1,mask)
Sout=gpio_Sout.read()
print(bin(Sout))
# 第三个周期，输出Sout，验证三次输出是否为串行的110
gpio_CLK.write(0b0,mask)
gpio_Load.write(0b0,mask)
gpio_Sin.write(0b0,mask)
gpio_D.write(0b000,mask1)
gpio_CLK.write(0b1,mask)
Sout=gpio_Sout.read()
print(bin(Sout))
```

3）源代码

本实验源代码中所含文件介绍同8.1.1节。

4. 实验步骤

实验步骤同8.1.1节。

8.1.6 计数器

1. 实验目的

了解三位二进制计数器的工作原理，实现其功能，并完成十进制计数器的设计与实现。

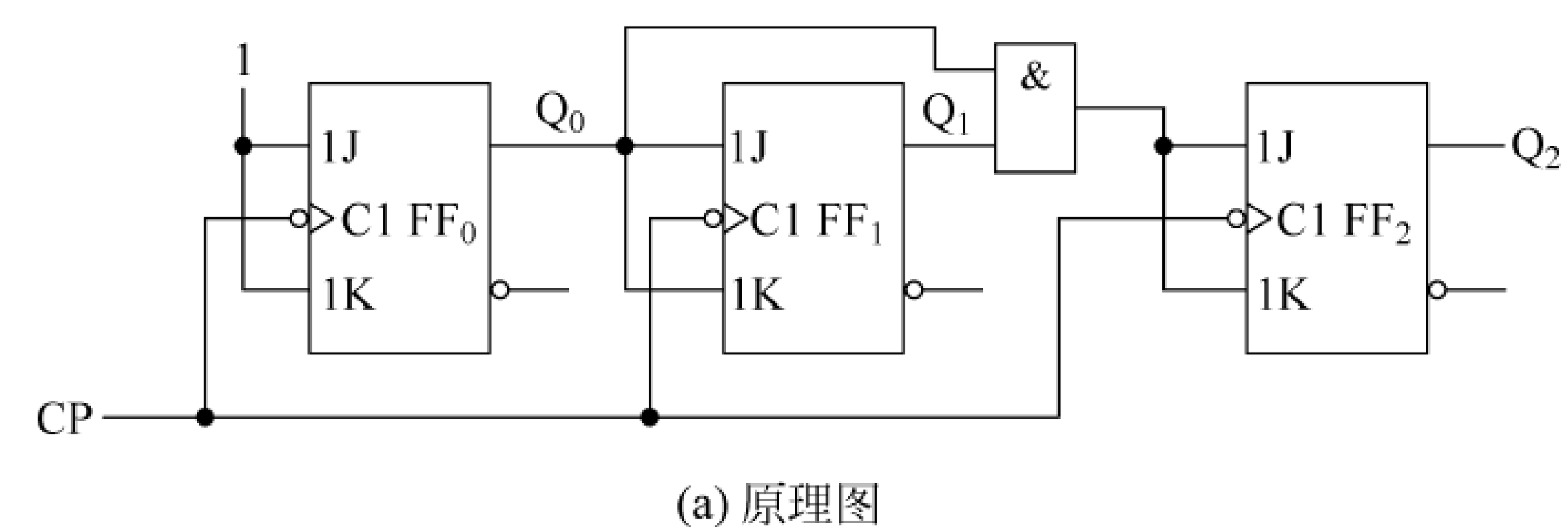
2. 实验要求

要求采用源代码中对应目录下的“基础模块”所提供的“.v”源文件生成IP，并采用原理图的方式调用所生成的IP设计所要求的系统。采用Vivado进行仿真，并进行I/O的连接，最后生成比特流文件，并下板完成验证。

3. 实验说明

1）实验原理

三位二进制计数器原理图、逻辑表达式及对应的真值表如图8-12所示。通过以上原理图可以完成三位二进制计数器的设计，设计需要包含三个JK触发器和一个与门，JK触发器输入端J，K都为1时，输出端Q0会不断地在时钟上升沿进行0→1翻转，实现计数功能，通过理解三位二进制计数器，再视情况完成十进制计数器的设计实验，如图8-13所示。



(a) 原理图

$J_1=K_1=Q_0$

$J_2=K_2=Q_0Q_1$

…

$J_{n-1}=K_{n-1}=Q_0Q_1\cdots Q_{n-2}$

(b) 逻辑表达式

状态	$Q_2\ Q_1\ Q_0$
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

(c) 真值表

图8-12 三位二进制计数器

2）Python交互模板

（1）加载比特流代码：

```
from pynq import Overlay                                          #调用Python库
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/......bit")     #bit文件路径
overlay?
```

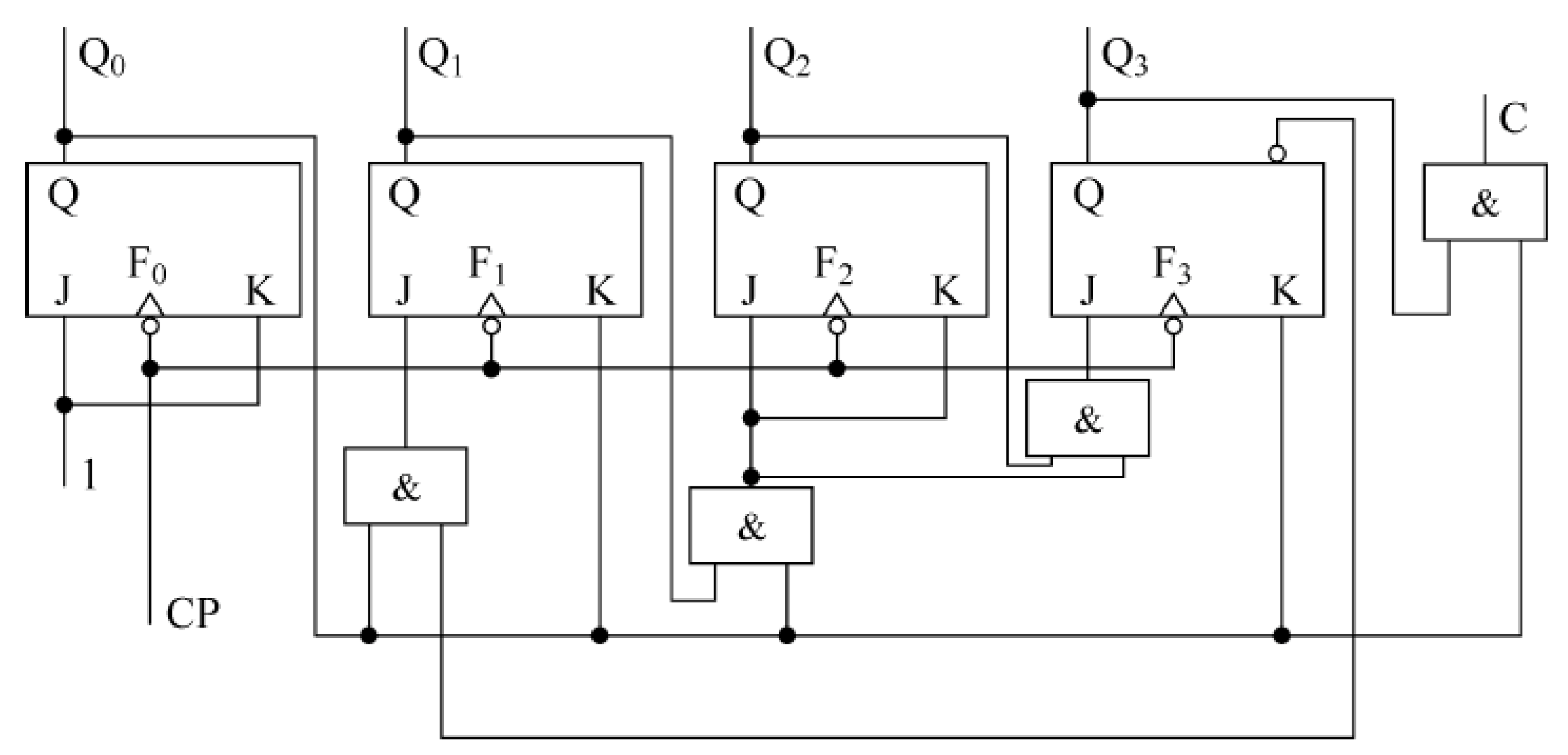


图 8-13 十进制计数器原理图

(2) GPIO 口的命名代码:

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_2 = overlay.ip_dict['axi_gpio_2']
gpio_3 = overlay.ip_dict['axi_gpio_3']

set_n = AxiGPIO(gpio_1).channel2              #置位信号
jk= AxiGPIO(gpio_0).channel2                  #j,k两个端口的输入数值(为1)
q0 = AxiGPIO(gpio_2).channel1                 #输出q0
q1 = AxiGPIO(gpio_2).channel2
q2 = AxiGPIO(gpio_3).channel1

clk = AxiGPIO(gpio_0).channel1                #时钟信号
rst_n = AxiGPIO(gpio_1).channel1              #复位信号
```

(3) GPIO 口的读写,代码如下:

```
import time                                   #调用事件函数
mask=0b1
timedelay = 0.1                               #设置时间延时
rst_n.write(0b0,mask)                         #启动复位
rst_n.write(0b1,mask)                         #停止复位

import time
N = 20                                        #设置循环次数
timedelay = 0.1
sizes = range(0,N,1)                          #设置循环且步长为1
mask=0b1
jk.write(0b1, mask)                           #jk端口输入处初值
set_n.write(0b1, mask)                        #set_n信号置1
tables = []
for size in sizes:
    clk.write(0b0,mask)
    clk.write(0b1,mask)
#通过Jupyter来生成时钟, 包含时序的电路通过jupyter来生成时钟避免出现时序的错误

    tmp=[]                                    #建立一个列表


    a2 =q2.read()
    tmp.append(a2)                            #将读出的数据写入到列表中

    a1 =q1.read()
    tmp.append(a1)

    a0 =q0.read()
    tmp.append(a0)

    print(tmp)                                #打印列表
```

3）源代码

本实验源代码中所含文件介绍同 8.1.1 节。

4. 实验步骤

实验步骤同 8.1.1 节。

8.1.7 有限状态机

1. 实验目的

学习三段式状态机的基本构成和工作原理，实现三段式状态机并对其加以改进和优化。

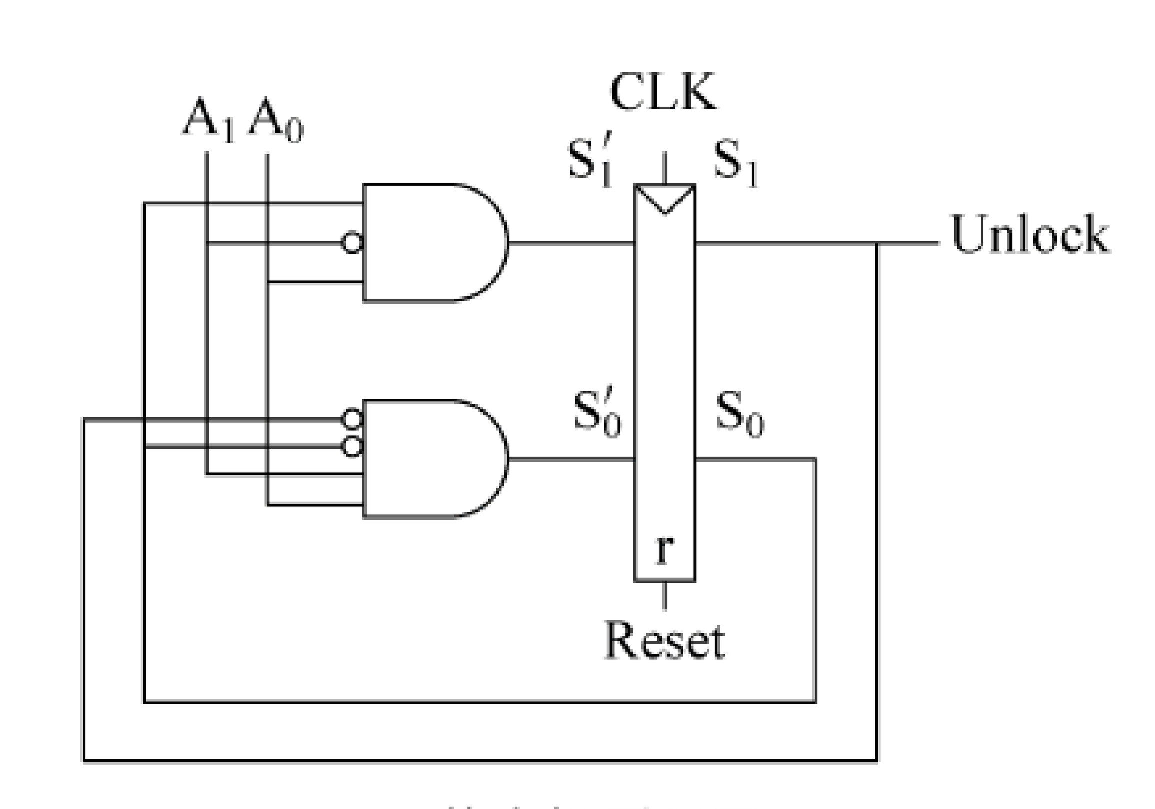
2. 实验要求

要求采用源代码中对应目录下的“基础模块”所提供的“.v”源文件生成 IP，并采用原理图的方式调用所生成的 IP 设计所要求的系统。采用 Vivado 进行仿真，并进行 I/O 的连接，最后生成比特流文件，并下板完成验证。

3. 实验说明

1）实验原理

状态机的原理图、简化的下一个状态表和简化的输出表如图 8-14 所示。当输入特定值时，状态机的状态也会发生改变，简化的状态转换表如图(b)所示。此状态机为一个简易密码锁，如果输入的 S_0 和 S_1 为 11，则状态机会从状态 0 进入状态 1；如果继续输入的数值为 01，则会进入状态 2 并打开锁；如果继续输入的不是 01，则回到状态 0。



(a) 状态机原理图

当前状态		输 入		下一个状态	
S_1	S_2	A_1	A_0	S_1	S_2
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	1	1	0
1	0	×	×	0	0

(b) 简化状态表

当前状态		输出
S_0	S_1	Y
0	0	0
0	1	0
1	0	1

(c) 简化的输出表

图 8-14 有限状态机

按上述的原理图连接各元件完成状态机实验，在用 Python 进行上班实验时需要用到图 8-14(b)状态表中 A_1 与 A_0 的数值，通过它们的数值变化来改变状态机的状态，当 S_0 与 S_1 为 1，0 时，Y 输出值为 1，否则都为 0。

2）Python 交互模板

（1）加载比特流的代码：

```
from pynq import Overlay                                        #调用Python库
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/......bit")    #bit文件路径
overlay?
```

（2）GPIO 口命名的代码：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_2 = overlay.ip_dict['axi_gpio_2']
gpio_3 = overlay.ip_dict['axi_gpio_3']
gpio_4 = overlay.ip_dict['axi_gpio_4']

A1 = AxiGPIO(gpio_0).channel1              #对输入输出端口进行命名
A0 = AxiGPIO(gpio_0).channel2

Y = AxiGPIO(gpio_1).channel1

clk = AxiGPIO(gpio_2).channel1
rst_n = AxiGPIO(gpio_2).channel2
```

（3）GPIO 口读写的代码：

```
import time                          #调用时间函数
mask=0b1
timedelay = 0.1                      #设置延时时间
rst_n.write(0b0,mask)
rst_n.write(0b1,mask)
```

```
import time
timedelay = 1
mask = 0b1

N =10                                #设置循环次数
sizes = range(0,N,1)
for size in sizes:
    if(size == 2):
        A1.write(0b0, mask)
        A0.write(0b0, mask)          #在第二次循环时设置A1和A0的数值
    elif(size == 3):
        A1.write(0b1, mask)
        A0.write(0b0, mask)
    elif(size == 5):
        A1.write(0b1, mask)
        A0.write(0b1, mask)
    elif(size == 6):
        A1.write(0b0, mask)
        A0.write(0b1, mask)
    tmp=[]                           #创建一个列表

    clk.write(0b0,mask)              #建立时钟
    clk.write(0b1,mask)

    a4 = Y.read()
    tmp.append(a4)

    print(tmp)
```

3）源代码

本实验源代码中所含文件介绍同 8.1.1 节。

4. 实验步骤

实验步骤同 8.1.1 节。

8.1.8 运算器/ALU

1. 实验目的

通过将 ALU 所需要的基本功能模块连接成完整的 ALU，理解 ALU 的功能和工作机制。本实验的 ALU 部分来自开源 RISC-V 处理器。

2. 实验要求

要求采用源代码中对应目录下的“基础模块”所提供的“.v”源文件生成 IP，并采用原理图的方式调用所生成的 IP 设计所要求的系统。采用 Vivado 进行仿真，并进行 I/O 的连接，最后生成流文件，并下板完成验证。

3. 实验说明

1）实验原理

ALU 的真值表如表 8-1 所列，普通 ALU 中包括上表给出的 12 种操作指令，每个 info 对应不同的指令类型，输入特定的 info 可以实现特定的指令操作，通过修改调试程序可以实现不同的计算功能。

表 8-1 ALU 真值表

指令类型	Info[31:0]	说明
LUI	0xc008	Imm<－12(左移 12 位)
SUB	0x28	rs1－rs2
ADD	0x18	rs1＋rs2
ADDI	0x8018	rs1＋imm
SLT	0x1008	if(rs1<rs2) return 1;else return 0
SLTI	0x9008	if(rs1<imm) return 1;else return 0
XOR	0x48	rs1^rs2(异或)
OR	0x408	rs1|rs2(或)
AND	0x808	rs1&rs2(与)
SLL	0x88	rs1 逻辑左移 rs2 位
SRL	0x108	rs1 逻辑右移 rs2 位
SRA	0x208	rs1 算术右移 rs2 位

实验内包含普通 ALU 模块和运算通路模块两个部分，其中普通 ALU 模块接收输入进来的 info 来判断是何种操作类型，并将输入进来的操作数和操作类型发送给运算通路模块，运算通路模块将接收到的操作数进行相应的运算，将结果返回给普通 ALU 模块。

2）Python 交互模板

(1) 加载比特流的代码：

```
from pynq import Overlay                                              #调用Python库
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/......bit")        #bit文件路径
overlay?
```

(2) GPIO 口命名的代码：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_rs1 = AxiGPIO(gpio_0).channel1                    #输入输出端口命名
gpio_rs2 = AxiGPIO(gpio_0).channel2
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_imm = AxiGPIO(gpio_1).channel1
gpio_info = AxiGPIO(gpio_1).channel2
gpio_2 = overlay.ip_dict['axi_gpio_2']
gpio_wdat = AxiGPIO(gpio_2).channel1
```

(3) GPIO 口读写的代码：

```
mask = 0xffffffff                                      #GPIO掩码
gpio_rs1.write(8,mask)                                 #写入数据
gpio_rs2.write(3,mask)
gpio_imm.write(0,mask)
gpio_info.write(0x18,mask)
wdat = gpio_wdat.read()
print(wdat)                                            #打印数据
```

3) 源代码

本实验源代码中所含文件介绍同 8.1.1 节。

4. 实验步骤

实验步骤同 8.1.1 节。

8.1.9　存储器

1. 实验目的

通过对基本存储器模块的使用，及使用多个存储器模块搭建更大的存储器系统，理解和掌握存储器的基本原理和用法。

2. 实验要求

根据提供的存储器基本模块，以字扩展(地址扩展)的方式把 2 片存储器连接成容量更大的存储器系统，完成仿真，并下板完成验证。

3. 实验说明

1) 实验原理

基础模块中包含两个独立的存储器单元，分别为 mem2 和 mem2_1，写入和读出数据时都需要选择存储器。通过片选可以控制要写入和读出的为哪一个存储器，片选信号能为 info 的第 0 和 1 位，当使用 mem2 时，info 最低的两位要使用数值 01 才可以正常地写入和读出；当使用 mem2_1 时，info 最低的两位为 10，如果片选信号错误，则存储器将不能正常地写入和读出数据。

2) Python 交互模板

(1) 加载比特流的代码：

```
from pynq import Overlay                                             #调用Python库
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/......bit")       #bit文件路径
overlay?
```

（2）GPIO 口命名的代码：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_2 = overlay.ip_dict['axi_gpio_2']

info1 = AxiGPIO(gpio_0).channel1                #输入输出端口命名
wdata1 = AxiGPIO(gpio_0).channel2
info2 = AxiGPIO(gpio_1).channel1
wdata2 = AxiGPIO(gpio_1).channel2

rdata1 = AxiGPIO(gpio_2).channel1
rdata2 = AxiGPIO(gpio_2).channel2
```

（3）GPIO 口读写的代码：

```
mask = 0xffffffff                               #GPIO掩码
info1.write(0x15, mask)                         #输入信息
wdata1.write(0x99999999, mask)                  #输入数据
info1.write(0x11, mask)
out1 = hex(rdata1.read())
print('out1 =',out1)                            #打印数据

info2.write(0x26, mask)
wdata2.write(0x88888888, mask)
info2.write(0x22, mask)
out2 = hex(rdata2.read())
print('out2 =',out2)
```

3）源代码

本实验源代码中所含文件介绍同 8.1.1 节。

4. 实验步骤

实验步骤同 8.1.1 节。

8.1.10 寄存器堆

1. 实验目的

了解寄存器的基本构成和工作原理，能够在单个寄存器基础上搭建寄存器组，实现具有 5 个寄存器的数据通路。

2. 实验要求

要求采用源代码中对应目录下的“基础模块”所提供的“.v”源文件生成 IP，并采用原理图的方式调用所生成的 IP 设计所要求的系统。采用 Vivado 进行仿真，并进行 I/O 的连接，最后生成比特流文件，并下板完成验证。

3. 实验说明

1）实验原理

如图 8-15 所示，数据写入时默认暂存到寄存器 DR 中，DR 的输出可以更新 R0～R4 中的任意一个寄存器，更新到具体哪一个寄存器需要由各个寄存器的使能信号决定，当某个寄存器使能为

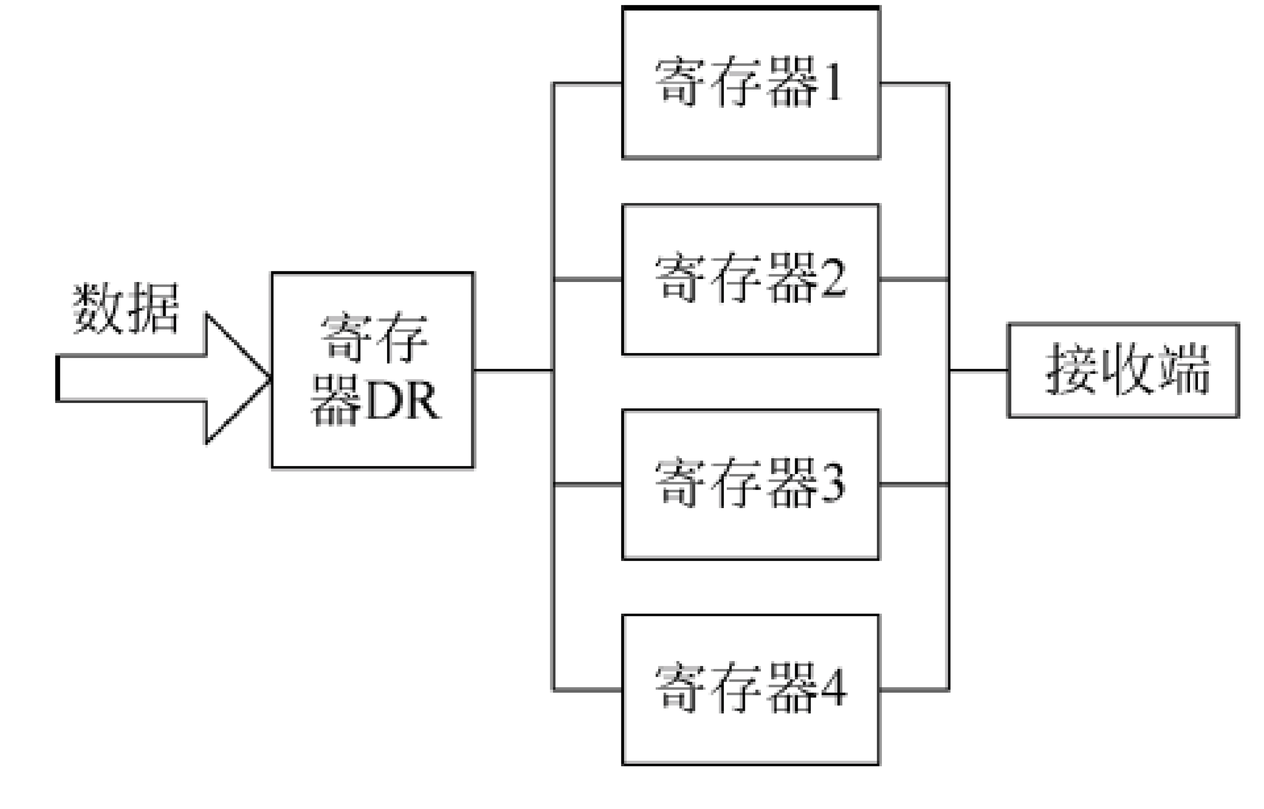


图 8-15 寄存器组示意图

1 时，寄存器 DR 中的数据进入相应的寄存器中，接收到数据的寄存器将数据暂存，并将数据发送到接收端以供用户观察实验效果。

2）Python 交互模板

（1）加载比特流的代码：

```
from pynq import Overlay                                                    #调用Python库
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/......bit")     #bit文件路径
overlay?
```

（2）GPIO 口命名的代码：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_2 = overlay.ip_dict['axi_gpio_2']
gpio_3 = overlay.ip_dict['axi_gpio_3']

read_idx1 = AxiGPIO(gpio_0).channel1          #输入输出端口命名
read_idx2 = AxiGPIO(gpio_0).channel2
w_idx = AxiGPIO(gpio_1).channel1
w_data = AxiGPIO(gpio_1).channel2

read_data1 = AxiGPIO(gpio_2).channel1
read_data2 = AxiGPIO(gpio_2).channel2
```

（3）GPIO 口读写的代码：

```
mask = 0xffffffff                             #GPIO掩码
w_idx.write(0x00000001, mask)                 #写入数据
w_data.write(0xf0f0f0f0, mask)
read_idx2.write(0x00000001, mask)

out = hex(read_data2.read())                  #转换成16位数据
print(out)                                    #打印数据
```

3）源代码

本实验源代码中所含文件介绍同 8.1.1 节。

4. 实验步骤

实验步骤同 8.1.1 节。

8.1.11　总线

1. 实验目的

了解总线的基本构成和工作原理，实现基于总线进行数据传输的寄存器数据写入与读取。

2. 实验要求

要求采用源代码中对应目录下的“基础模块”所提供的“.v”源文件生成 IP，并采用原理图的方式调用所生成的 IP 设计所要求的系统。采用 Vivado 进行仿真，并进行 I/O 的连接，最后生成比特流文件，并下板完成验证。

3. 实验说明

1）实验原理

本实验采用蜂鸟 E203 处理器中自定义的总线协议 ICB(Internal Chip Bus)。该总线应

用于蜂鸟 E203 处理器中，其结合 AXI 总线和 AHB 总线的优点，兼具高速性和易用性。其总线通道结构如图 8-16 所示。

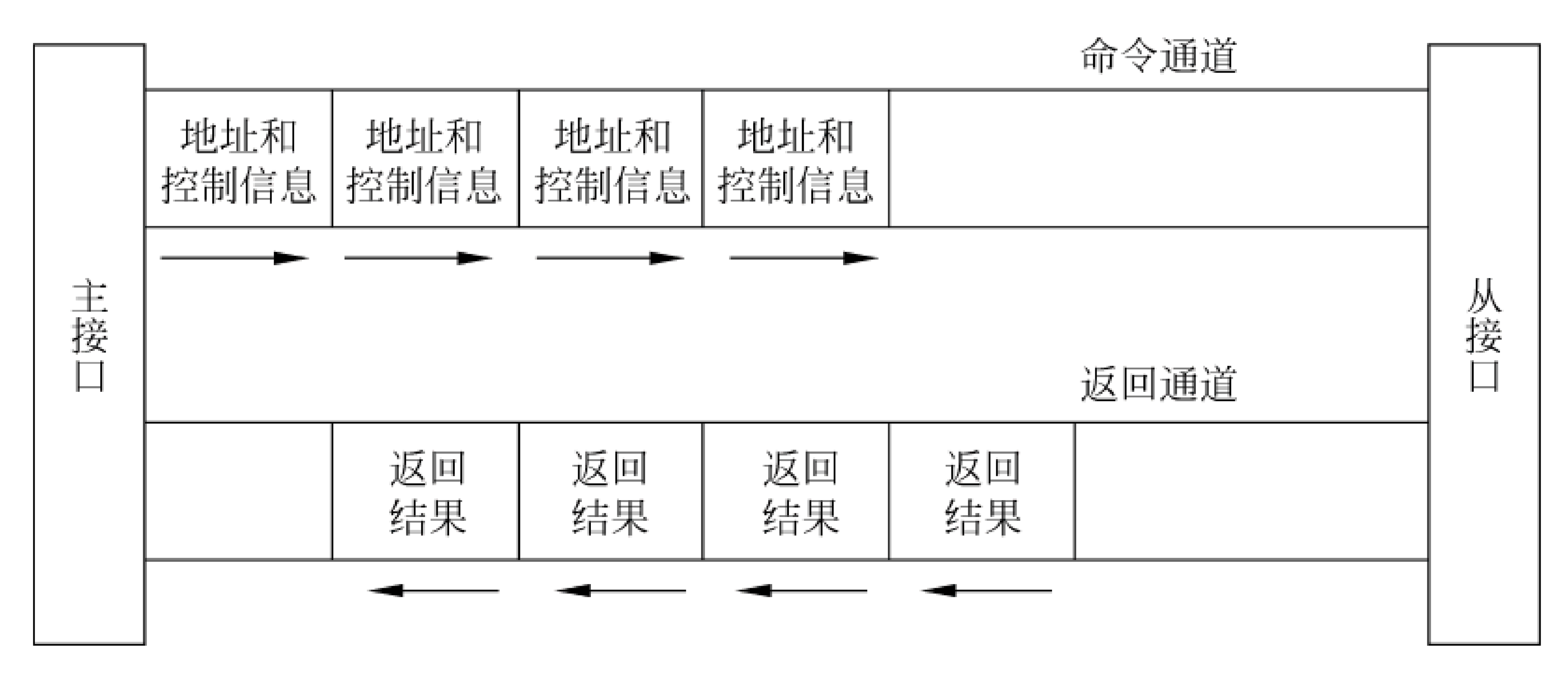


图 8-16　总线通道结构

表 8-2 所示为 ICB 总线的参数。

表 8-2　ICB 总线信号

通道	方向	宽度	信号名	介　绍
命令通道	Output	1	Icb_cmd_valid	主设备向从设备发送读写请求信号
	Input	1	Icb_cmd_ready	从设备向主设备返回读写接收信号
	Output	8	Icb_cmd_addr	读写地址
	Output	1	Icb_cmd_read	读写指示
	Output	16	Icb_cmd_wdata	写操作的数据
	Output	16	Icb_cmd_wmask	写操作时的字节掩码
返回通道	Input	1	Icb_rsp_valid	从设备向主设备发送读写反馈请求信号
	Output	1	Icb_rsp_ready	主设备向从设备返回读写反馈接收信号
	Input	16	Icb_rsp_rdata	读反馈的数据
	Input	1	Icb_rsp_err	读或写反馈的错误标志

表中：命令通道(Command Channel)主要用于主设备向从设备发起读写请求；返回通道(Response Channel)主要用于从设备向主设备返回读写结果。

ICB 总线时序有若干种，图 8-17 为典型时序中的其中一种：主设备向从设备通过 ICB 的 Command Channel 发送写操作请求(icb_cmd_read 为低)，从设备立即接收该请求(icb_cmd_ready 为高)。从设备在同一个周期返回读结果且结果正确(icb_rsp_err 为低)，主设备立即接收该结果(icb_rsp_ready 为高)。

源代码中包含总线模块和 ram 模块两个部分，其中 PYNQ 的 PS 端为主模块，ram 模块为从模块，用户通过 PS 端发送数据，总线模块接收数据并将数据传入 ram 模块中，完成本实验后可以根据学习情况根据 ICB 总线协议自行设计一个从模块并进行验证。

2）Python 交互模板

(1) 加载比特流的代码：

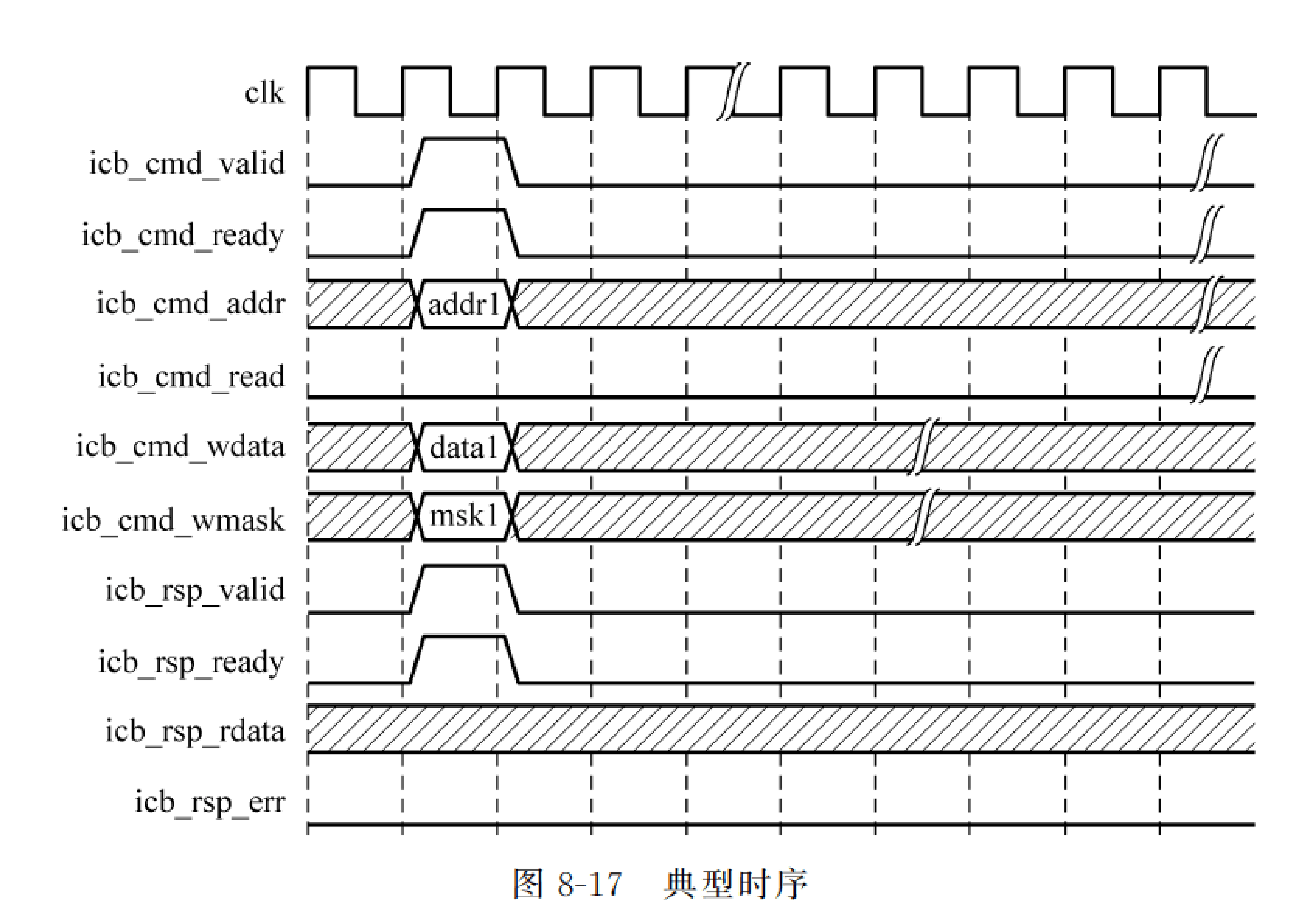


图 8-17 典型时序

```
from pynq import Overlay                                              #调用Python库
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/......bit")        #bit文件路径
overlay?
```

（2）GPIO 口命名的代码：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_2 = overlay.ip_dict['axi_gpio_2']
gpio_3 = overlay.ip_dict['axi_gpio_3']
gpio_4 = overlay.ip_dict['axi_gpio_4']
gpio_5 = overlay.ip_dict['axi_gpio_5']

ifu2biu_cmd_valid = AxiGPIO(gpio_0).channel1        #对端口进行命名
ifu2biu_cmd_addr = AxiGPIO(gpio_0).channel2
ifu2biu_cmd_read = AxiGPIO(gpio_1).channel1
ifu2biu_cmd_wdata = AxiGPIO(gpio_1).channel2
ifu2biu_cmd_wmask = AxiGPIO(gpio_2).channel1
ifu2biu_rsp_ready = AxiGPIO(gpio_2).channel2
mem_icb_enable = AxiGPIO(gpio_3).channel1
icb2biu_cmd_ready = AxiGPIO(gpio_3).channel2
ifu2biu_rsp_valid = AxiGPIO(gpio_4).channel1
ifu2biu_rsp_err = AxiGPIO(gpio_4).channel2
ifu2biu_rsp_rdata = AxiGPIO(gpio_5).channel1
```

（3）GPIO 口读写的代码：

```
mask = 0xffff                                      #GPIO输入掩码
ifu2biu_cmd_valid.write(0b1,mask)                  #对各端口进行赋值
ifu2biu_cmd_addr.write(0x01,mask)
ifu2biu_cmd_read.write(0b0,mask)
ifu2biu_rsp_ready.write(0b1,mask)
ifu2biu_cmd_wdata.write(0xeeee,mask)
ifu2biu_cmd_wmask.write(0xffff,mask)
mem_icb_enable.write(0b1,mask)
```

```
mask = 0xffff
ifu2biu_cmd_read.write(0b1,mask)                #读出输出数据
ifu2biu_cmd_addr.write(0x01,mask)

print(ifu2biu_rsp_err.read())                   #打印数据
rdata = hex(ifu2biu_rsp_rdata.read())
print(rdata)
```

3）源代码

本实验源代码中所含文件介绍同 8.1.1 节。

4. 实验步骤

实验步骤同 8.1.1 节。

8.1.12 微程序控制器

1. 实验目的

掌握基于 PYNQ 平台及 Vivado 工具的时序逻辑电路的开发流程及调试方式。本实验提供了一个简单的 CPU 模型机，要求学生在理解其工作原理的基础上，补充完善微程序控制器中微程序的设计，使其可以仿真并下板运行。

2. 实验要求

要求采用源代码中对应目录下的“基础模块”所提供的“.v”源文件生成 IP，并采用原理图的方式调用所生成的 IP 设计所要求的系统。采用 Vivado 进行仿真，并进行 I/O 的连接，最后生成比特流文件，并下板完成验证。

3. 实验说明

1）实验原理

（1）CPU 数据通路：如图 8-18 所示，每一条机器指令的执行过程都可分为取指和执行两个阶段。

程序计数器(PC)是一条指令的起点，所以在取指阶段，首先由 PC 输出将要执行的机器指令的地址，接着指令存储器(Imem)将与该地址对应的机器指令输出到指令寄存器(IR)，同时 PC 指向下一条要执行的指令，为下次取指做准备。

接着，IR 将指令的操作码送入指令译码器(Decoder)，译码器中存储有与之对应的微程序的入口地址，将其送入到微程序计数器(MPC)，最终由 MPC 确定将要执行的微指令地址，将该条微指令从微指令控制存储器(CUmem)中读出，将这些控制信号发送送到 CPU 的各个组件，指令便开始执行。

（2）指令设计：该模型机设计了 3 条数据转移类指令(MOV，MOVI，LAD，LADI，STO)，7 条运算类指令(ADD，SUB，INC，DEC，AND，OR，NOT)，7 条跳转类指令(JEQ，JNE，JHI，JHS，JLO，JLS，JMP)以及比较指令(CMP)和停机指令(HLT)，共 21 条指令。其中，仅有 LAD 与 STO 可以访问存储器，大多数的指令采用寄存器寻址，LAD 和 STO 的寻址方式为间接寻址，跳转指令及部分第二操作数为立即数的指令采用立即寻址方式。

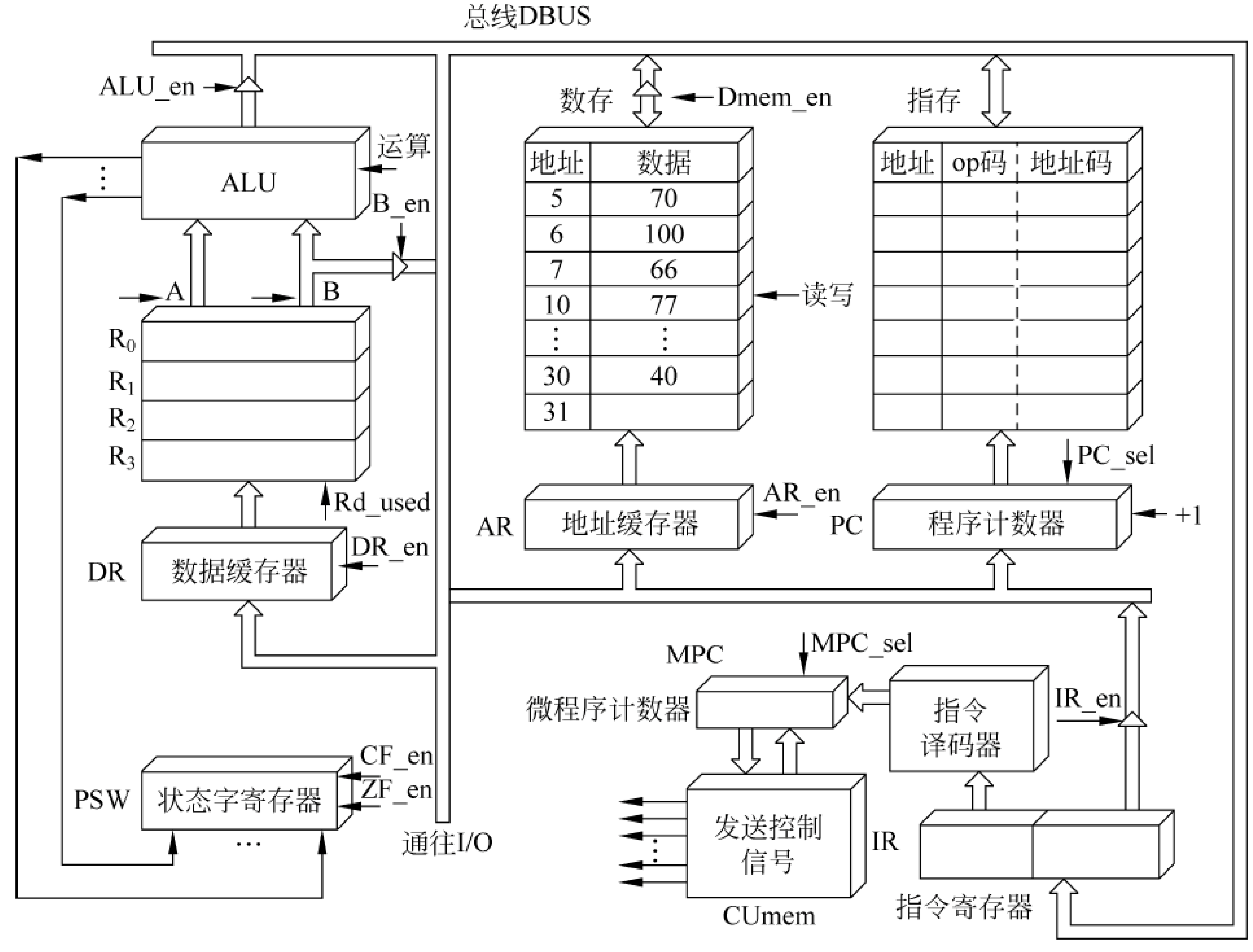


图 8-18　CPU 数据通路图

具体指令设计请阅读源代码资料包中本实验文件夹中的“附录 2”。

(3) 取指微指令的实现：

第一条微指令：

ZF_en	PC_en	PC_sel	Imem_en	MPC_en	MPC_sel	cs	we
0	0	0	1	1	10	0	0
ALU_en	B_en	Dmem_en	IR_data_en	CF_en	Rd_used	RsA_used	RsB_used
0	0	0	0	0	0	0	0
AR_en	outenab	M	S(4 位)		Next_addr(16 位)		
0	0	0	0000		0000 0000 0000 0001		

① 第一个周期：PC 中的地址送入指令寄存器(Imem)，通过发送控制信号 Imem_en，使该地址对应的指令操作码(Opcode)从 Imem 经过总线 DBUS 进入指令译码器，经过翻译，从指令译码器流出的数据变成了这条指令在 CUmem 中的入口地址，入口地址被输入到微程序计数器 MPC 的 MPC_in 中。同时发送了控制信号 MPC_sel＝“10”，代表下一条微指令地址将从 CU_mem 传回的 next_addr 中获取(此处 next_addr 即为取指程序的第二条微指令地址)。

第二条微指令：

ZF_en	PC_en	PC_sel	Imem_en	MPC_en	MPC_sel	cs	we
0	1	0	0	1	11	0	0
ALU_en	B_en	Dmem_en	IR_data_en	CF_en	Rd_used	RsA_used	RsB_used
0	0	0	0	0	0	0	0
AR_en	outenab	M	S(4 位)		Next_addr(16 位)		
0	0	0	0000		00 0000 0000 0000		

② 第二个周期：由于控制单元发送的微程序计数器(MPC)的选择信号 MPC_sel＝“11”(意为将 MPC_in 输出)，指令的入口地址便被送入 CUmem，CUmem 便从该指令入口地址开始，逐条发送该指令对应的控制信号。同时，可以注意到表中的第二行，PC_en 被置为“1”，且 PC_sel＝“00”(意为 PC＝PC＋1)，从而使 PC 指向下一条指令，为下一次取指做好准备。

(4) 代码层次结构如下：

```
|-- tinycpu.v
    |-- ALU.v
    |-- DBUS.v
    |-- Regsfile.v
    |-- Dmem.v
    |-- Imem.v
    |-- Program_Counter.v
    |-- Decoder.v
    |-- MPC.v
    |-- CUmem.v
    |-- psw.v
```

具体代码请阅读基础模块文件夹。

2) Python 交互模板

(1) 加载比特流的代码：

```
from pynq import Overlay
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/...|bit")
```

(2) GPIO 口命名的代码：

```
gpio_5 = overlay.ip_dict['axi_gpio_5']
gpio_m0 = AxiGPIO(gpio_5).channel1
gpio_m1 = AxiGPIO(gpio_5).channel2
gpio_6 = overlay.ip_dict['axi_gpio_6']
gpio_m2 = AxiGPIO(gpio_6).channel1
gpio_m3 = AxiGPIO(gpio_6).channel2
gpio_7 = overlay.ip_dict['axi_gpio_7']
gpio_m4 = AxiGPIO(gpio_7).channel1
gpio_m5 = AxiGPIO(gpio_7).channel2
```

（3）验证 CPU 是否成功排序的代码：

```
import time                                    #时间函数
timedelay = 0.0001                             #设置延时
time.sleep(timedelay )                         #睡眠
tmp=[]
m0 = gpio_m0.read()
tmp.append(m0)
m1 = gpio_m1.read()
tmp.append(m1)
m2 = gpio_m2.read()
tmp.append(m2)
m3 = gpio_m3.read()
tmp.append(m3)
m4 = gpio_m4.read()
tmp.append(m4)
m5 = gpio_m5.read()
tmp.append(m5)
print(tmp)
```

```
[0, 1, 4, 5, 6, 10]
```

3）源代码

本书配套源代码中，为实验提供了如下材料：

（1）“基础模块”文件夹：该文件夹下提供的是每个实验所需的基础模块“.v”源文件，在采用原理图方式进行实验时，用户可以将其生成 IP 备用。

（2）“test.v”：是对用户完成的设计进行仿真时所需的 Testbench 示例程序，用户可直接使用，也可以根据需要自行修改。

（3）“cpu.ipynb”是本节“Python 交互模板”部分所介绍的 Python 源文件。

（4）“附录 1”：AIU 运算表。

（5）“附录 2”：模型机的指令说明。

4. 实验步骤

（1）将 CUmem 中的微指令设计完整。

（2）使用本实验对应源代码中的 simu_cpu 仿真文件进行仿真，观察 m0～m5 的值是否为排序后的值（0，1，4，5，6，10）。因为仿真时间超过默认时间，所以需要在 simulation setting 中将 simulation runtime 设置为 40μs。

（3）将仿真后的设计生成原理图添加到 IPrepository。

（4）按照图 8-19 所示和接口模块相连接。

（5）重新综合、实现及生成比特流文件，并将比特流文件及对应 tcl 文件上传到所用的 PYNQ 节点。

（6）下板实验，使用本实验对应源代码中的 Python 调试文件，观察真实运行后的 CPU 模型机是否正常工作。

5. 实验建议

学生可以按照第 3 章中介绍的 HDL 开发流程，直接将本实验对应源代码中“基础模块”文件夹提供的源文件添加到工程中，无须再将每个模块生成原理图的方式。

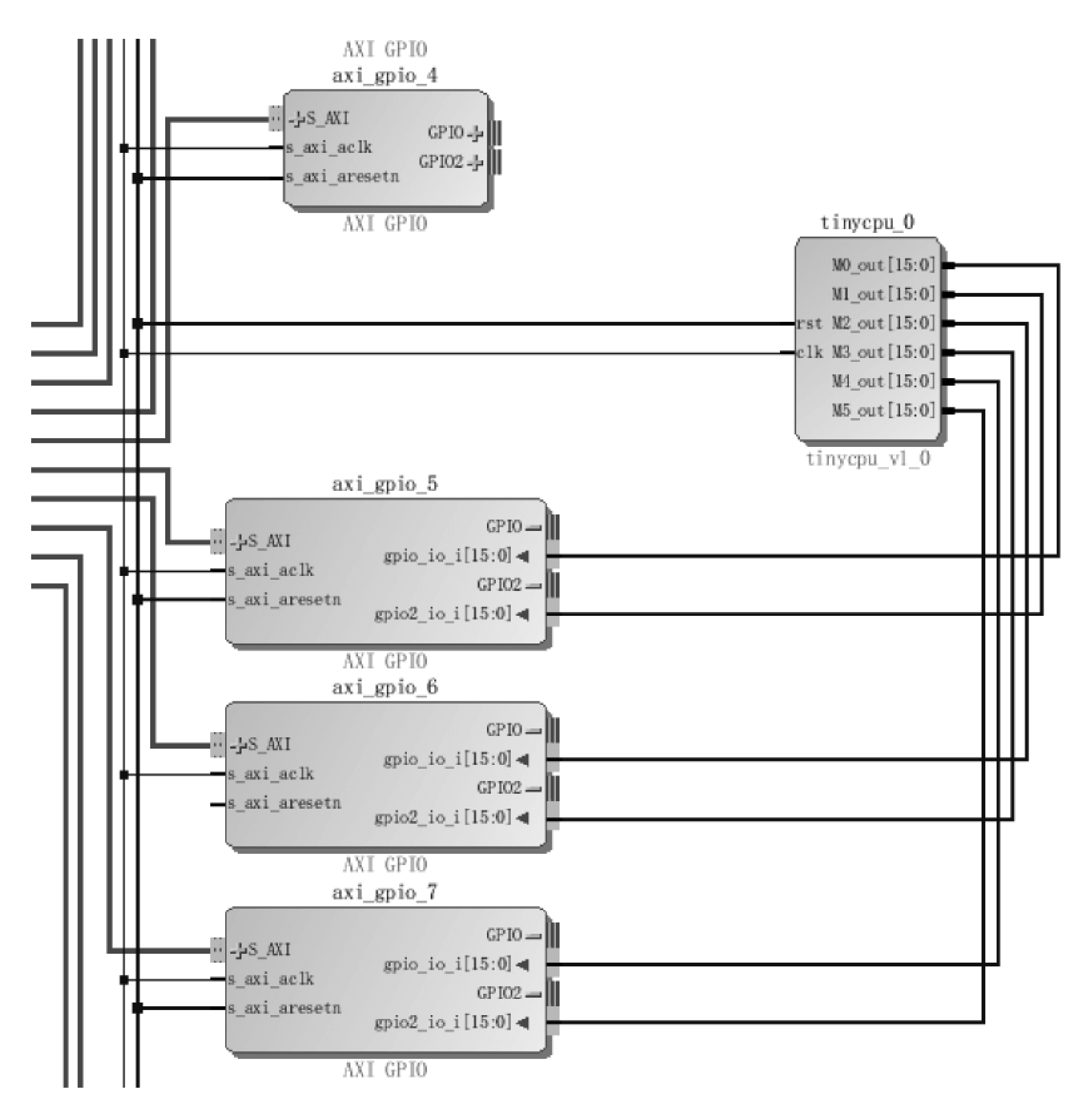


图 8-19　模型机与接口模块连接方式

8.2　基于 Verilog HDL 的实验

8.2.1　全加器

1. 实验目的

掌握基于 Verilog HDL 方式的 4 位全加器程序设计方法，通过 Vivado 软件和 PYNQ 平台进行开发调试，实现全加器的功能，并通过 4 位全加器程序完成 8 位全加器的实现。

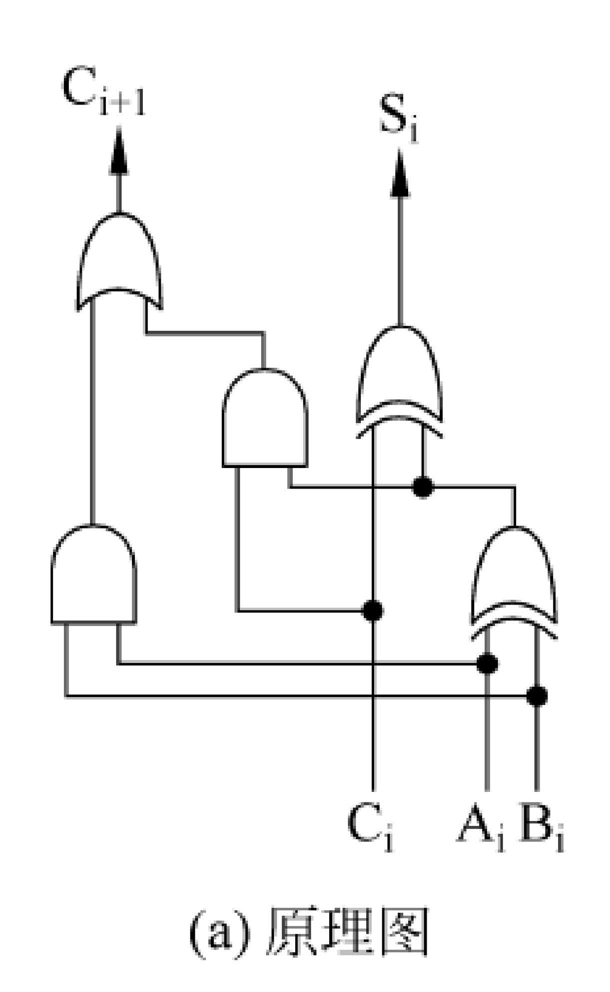
2. 实验要求

要求采用源代码中对应目录下的“基础模块”所提供的“.v”源文件建立完整的工程，使用其中的测试文件进行整体仿真，仿真完成后生成 IP，并调用所生成的 IP 进行 I/O 的连接，构成完整的系统。最后生成比特流文件，并下板完成验证。

3. 实验说明

1）实验原理

全加器的原理图、逻辑表达式及对应的真值表如图 8-20 所示。通过 Verilog HDL 语言可以直接编写全加器模块直接实现原理图的功能，具体全加器的代码可以参考源代码中对应目录下的“基础模块”所提供的“.v”源文件，通过理解代码并完成 8 位全加器的设计。



(a) 原理图

$S_i=A_i \oplus B_i \oplus C_i$
$C_{i+1}=A_iB_i+A_iC_i+B_iC_i$
$=A_iB_i+(A_i \oplus B_i)C_i$

(b) 逻辑表达式

输入			输出	
A_i	B_i	C_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(c) 真值表

图 8-20　一位全加器

2）Python 交互模板

（1）加载比特流的代码：

```
from pynq import Overlay                                              #调用Python库
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/......bit")        #bit文件路径
overlay?
```

（2）GPIO 口命名的代码：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_2 = overlay.ip_dict['axi_gpio_2']

a = AxiGPIO(gpio_0).channel1                  #输入输出端口命名
b = AxiGPIO(gpio_0).channel2
cin = AxiGPIO(gpio_1).channel1
s = AxiGPIO(gpio_1).channel2
cout = AxiGPIO(gpio_2).channel1
```

（3）加法器功能验证的代码：

```
mask = 0                                  #GPIO掩码
a.write(8,mask)                           #写入数据
b.write(9,mask)
cin.write(0,mask)

print(cout.read())                        #打印数据
print(s.read())
```

3）源代码

本书配套源代码中，为实验提供了如下材料：

（1）“基础模块”文件夹：该文件夹下提供的是每个实验所需的基础模块“.v”源文件，在进行实验时，用户可以将其生成 IP 直接使用。

（2）“test.v”：是对用户整体工程进行仿真时所需的 Testbench 示例程序，用户可直接使用，也可以根据需要自行修改。

(3)“xxx.ipynb”是本节“Python 交互模板”部分所介绍的 Python 源文件。

4. 实验步骤

(1) 熟悉基本模块的代码功能,可对基础模块进行更改,修改并使用本实验对应材料中的仿真文件进行仿真。

(2) 将本实验对应材料中提供的基本模块的“.v”文件生成 IP 核。

(3) 调用 IP 核,进行 I/O 处理,将系统的逻辑 I/O 与 PS 相连接,以便进行调试。

(4) 重新综合、实现及生成比特流文件,并将比特流文件及对应 tcl 文件上传到所用 PYNQ 节点。

(5) 下板实验,使用本实验对应材料中的 Python 调试文件,观察真实运行后的系统是否正常工作。

8.2.2 译码器

1. 实验目的

掌握基于 Verilog HDL 方式的二—四译码器程序设计方法,通过 Vivado 软件和 PYNQ 平台进行开发调试,实现二—四译码器的功能,在此基础上实现三—八译码器。

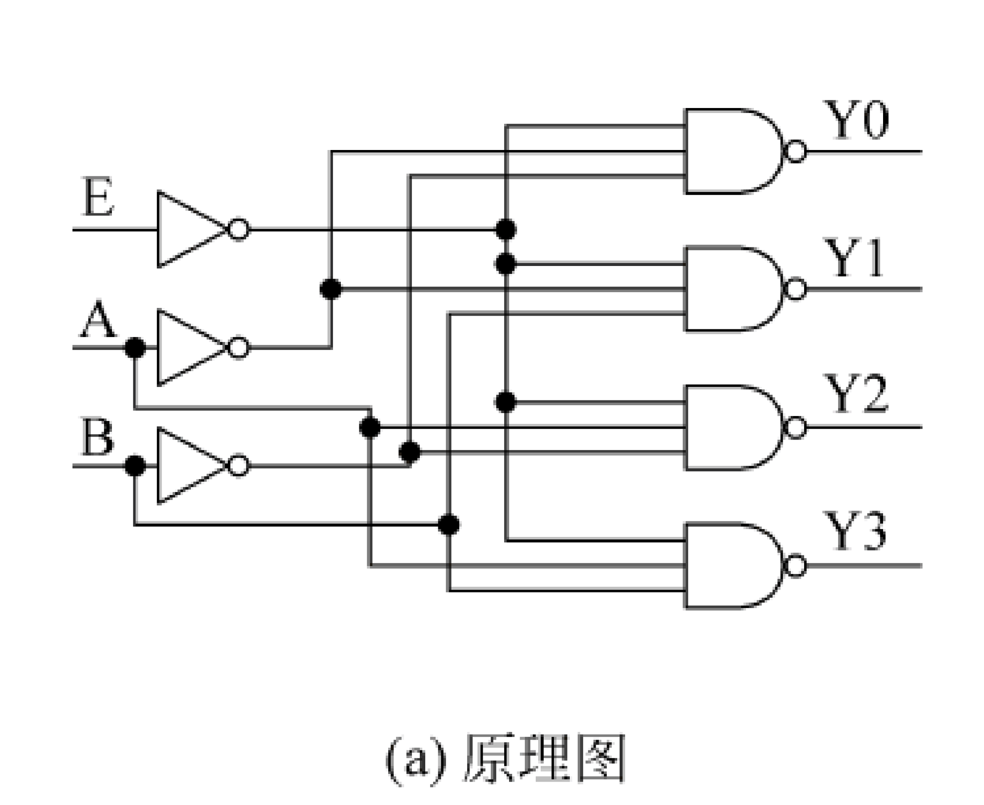
2. 实验要求

要求采用源代码中对应目录下的“基础模块”所提供的“.v”源文件建立完整的工程,使用其中的测试文件进行整体仿真,仿真完成后生成 IP,并调用所生成的 IP 进行 I/O 的连接,构成完整的系统。最后生成比特流文件,并下板完成验证。

3. 实验说明

1) 实验原理

二—四译码器的原理图及对应的真值表如图 8-21 所示。通过 Verilog HDL 语言可以直接编写二—四译码器直接实现原理图的功能,具体译码器的代码可以参考源代码中对应目录下的“基础模块”所提供的“.v”源文件,通过理解代码并完成三—八译码器的设计。



(a) 原理图

输入			输出			
E	A	B	Y0	Y1	Y2	Y3
1	×	×	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

(b) 真值表

图 8-21 二—四译码器

2) Python 交互模板

(1) 加载比特流的代码:

```
from pynq import Overlay                                              #调用Python库
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/......bit")       #bit文件路径
overlay?
```

(2) GPIO 口的定义与原理图的连接一致,其代码如下:

```
gpio_0 = overlay.ip_dict['axi_gpio_0']      #输入输出端口命名
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_2 = overlay.ip_dict['axi_gpio_2']

a = AxiGPIO(gpio_0).channel1
y = AxiGPIO(gpio_0).channel2
```

(3) GPIO 口读写的代码:

```
mask = 0xf                                  #GPIO掩码
a.write(3,mask)
out = bin(y.read())                         #打印数据
print(out)                                  #打印数据
```

3) 源代码

本书配套源代码中,为实验提供了如下材料:

(1) "基础模块"文件夹:该文件夹下提供的是每个实验所需的基础模块".v"源文件,在进行实验时,用户可以将其生成 IP 直接使用。

(2) "test.v":是对用户整体工程进行仿真时所需的 Testbench 示例程序,用户可直接使用,也可以根据需要自行修改。

(3) "xxx.ipynb"是本节"Python 交互模板"部分所介绍的 Python 源文件。

4. 实验步骤

(1) 熟悉基本模块的代码功能,可对基础模块进行更改,修改并使用本实验对应材料中的仿真文件进行仿真。

(2) 将本实验对应材料中提供的基本模块的".v"文件生成 IP 核。

(3) 调用 IP 核,进行 I/O 处理,将系统的逻辑 I/O 与 PS 相连接,以便进行调试。

(4) 重新综合、实现及生成流文件,并将比特流文件及对应 tcl 文件上传到所用 PYNQ 节点。

(5) 下板实验,使用本实验对应材料中的 Python 调试文件,观察真实运行后的系统是否正常工作。

8.2.3 多路选择器

1. 实验目的

掌握基于 Verilog HDL 方式的四选一多路选择器程序设计方法,通过 Vivado 软件和 PYNQ 平台进行开发调试,实现四选一多路选择器的功能,在此基础上实现八选一译码器的功能。

2. 实验要求

要求采用源代码中对应目录下的"基础模块"所提供的".v"源文件建立完整的工程,使用其中的测试文件进行整体仿真,仿真完成后生成 IP,并调用所生成的 IP 进行 I/O 的连接,构成完整的系统。最后生成比特流文件,并下板完成验证。

3. 实验说明

1) 实验原理

通过 Verilog HDL 语言可以直接编写四选一多路选择器来实现如原理图实验相同的

功能，四选一多路选择器的代码可以参考源代码中对应目录下的“基础模块”所提供的“.v”源文件，通过理解代码并完成八选一多路选择器的设计。四选一多路选择器电路符号如图 8-22 所示。

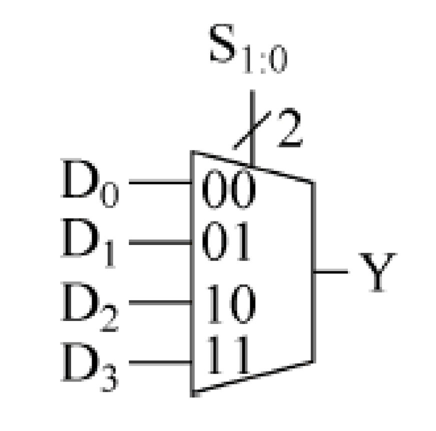


图 8-22 四选一多路选择器

2）Python 交互模板

（1）加载比特流的代码：

```
from pynq import Overlay                                          #调用Python库
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/......bit")     #bit文件路径
overlay?
```

（2）GPIO 口命名的代码：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_2 = overlay.ip_dict['axi_gpio_2']

select0 = AxiGPIO(gpio_0).channel1                  #输入输出端口命名
out0 = AxiGPIO(gpio_0).channel2
```

（3）GPIO 口读写的代码：

```
mask = 0xf                                          #GPIO掩码
select0.write(3,mask)                               #写入数据
data = hex(out0.read())                             #转换成16位数据
print(data)                                         #打印数据
```

3）源代码

本实验源代码中所含文件介绍同 8.2.2 节。

4. 实验步骤

实验步骤同 8.2.2 节。

8.2.4 触发器与寄存器

1. 实验目的

掌握基于 Verilog HDL 方式的 1 位 D 触发器和 8 位寄存器程序设计方法，通过 Vivado 软件和 PYNQ 平台进行开发调试，实现 1 位 D 触发器和 8 位寄存器程序的功能，能够在 D 触发器的基础上进一步设计带使能端和同步复位功能的触发器和带使能端和同步复位功能的 8 位寄存器的实验。

2. 实验要求

要求采用源代码中对应目录下的“基础模块”所提供的“.v”源文件建立完整的工程，使用其中的测试文件进行整体仿真，仿真完成后生成 IP，并调用所生成的 IP 进行 I/O 的连接，构成完整的系统。最后生成比特流文件，并下板完成验证。

3. 实验说明

1）实验原理

（1）D 触发器：一个 D 触发器可以由反相时钟控制的两个背靠背的 D 锁存器构成，其简化的电路符号如图 8-23 所示。D 触发器在时钟

图 8-23 D 触发器

上升沿将输入传给输出，在其他时间 D 触发器保持原来的状态。

寄存器中的一位用简单的同步 D 触发器就可以实现，8 位的寄存器是用了 8 个同步 D 触发器构成。

（2）带使能端的触发器：带使能端的触发器增加一个称为 EN 的输入，该输入用于确定时钟沿是否载入数据。当 EN=1 时，带使能端的触发器与普通 D 触发器一样；当 EN=0 时，带使能端的触发器忽略时钟，保持原来的状态；当希望在某些时钟沿而不是所有时钟沿载入一个新值时，带使能端的触发器就派上了用场。

注意：不要在时钟上执行逻辑（在时钟上改动），因为这样可能会使时钟延时并导致时序错误。

（3）带同步复位功能的触发器：带同步复位功能的触发器增加了一个称为 RESET 的输入。RESET 是一个低电平有效的信号，即当其为 0 时执行复位功能。当 RESET=0 时，带同步复位功能的触发器忽略输入 D 并在时钟上升沿将输出 Q 复位为 0；当 RESET=1 时，带同步复位功能的触发器与普通 D 触发器一样。

（4）带使能端和同步复位功能的触发器：带使能端和同步复位功能的触发器增加了两个输入 EN 和 RESET。EN 的优先级比 RESET 高，即 EN 为 1 时，RESET 无效。

2）Python 交互模板

（1）加载比特流的代码：

```
from pynq import Overlay                                                    #调用Python库
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/......bit")      #bit文件路径
overlay?
```

（2）GPIO 口命名的代码：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_2 = overlay.ip_dict['axi_gpio_2']

din = AxiGPIO(gpio_0).channel1                  #输入输出端口命名
dout = AxiGPIO(gpio_0).channel2
```

（3）GPIO 口读写的代码：

```
mask = 0xff                                     #GPIO掩码
din.write(0xaa,mask)                            #写入数据
data = hex(dout.read())                         #转换成16位数据
print(data)                                     #打印数据
```

3）源代码

本实验源代码中所含文件介绍同 8.2.2 节。

4. 实验步骤

实验步骤同 8.2.2 节。

8.2.5　移位寄存器

1. 实验目的

掌握基于 PYNQ 平台及 Vivado 工具的时序逻辑电路的开发流程及调试方式，了解移

位寄存器的 Verilog 行为描述，能够在此基础上使用 Verilog 描述 8 位的带并行加载的移位寄存器。

2. 实验要求

要求使用 Verilog 行为描述完成目标系统的设计，采用 Vivado 进行仿真，并进行 I/O 的连接，最后生成比特流文件，并下板完成验证。

3. 实验要求

1）实验原理

（1）移位寄存器：移位寄存器的原理图如图 8-24 所示，它由 N 个触发器串联而成。移位寄存器包括时钟、串行输入 S_{in}、串行输出 S_{out} 和 N 位并行输出 $Q_{N-1:0}$。在时钟的每一个上升沿，从 S_{in} 移入一个个新的位，所有后续内容都向前移动。移位寄存器可以完成串行到并行的转换。输入由 S_{in} 以串行方式提供，在 N 个周期后，前面的 N 位输入可以在 Q 中并行访问。在此基础上增加一个低电平异步复位信号 reset，Verilog 的行为描述请参考“基础模块”的 shift_reg. v 文件。

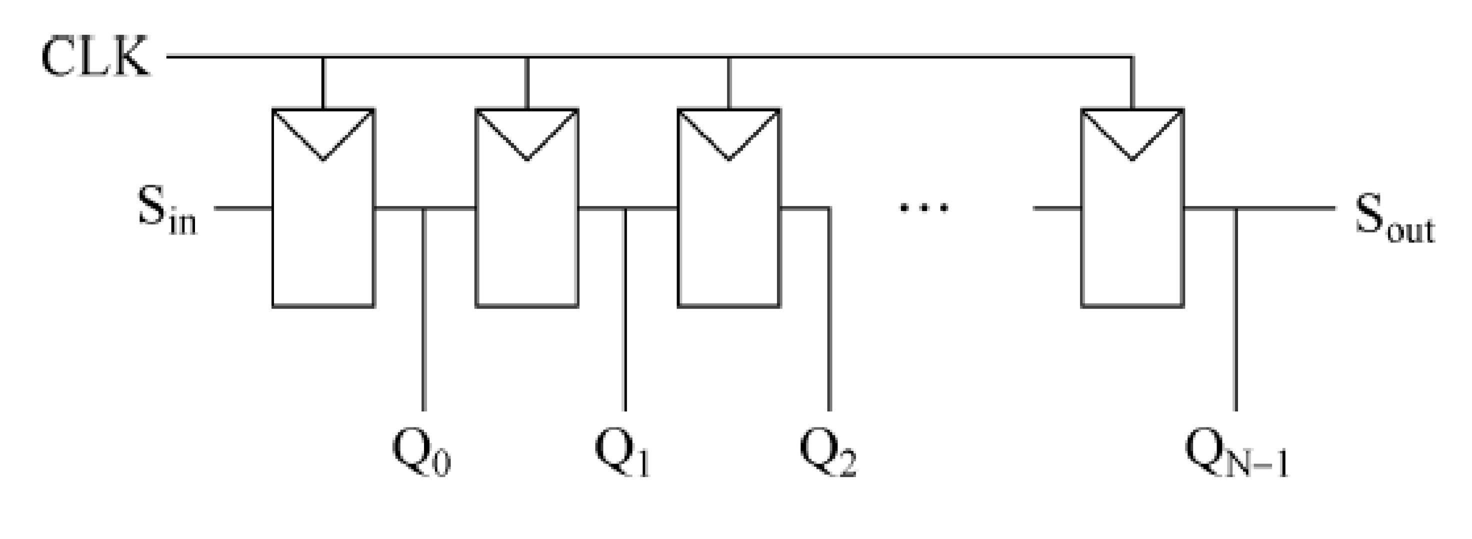


图 8-24　移位寄存器

（2）带并行加载的移位寄存器：带并行加载的移位寄存器增加了并行输入 $D_{N-1:0}$ 和控制信号 Load，它既可以完成串行到并行的转换，也可以完成并行到串行的转换。当 Load 有效时，触发器从输入 D 中并行加载数据，否则移位寄存器正常移位，在 N 个周期后就将 $D_{N-1:0}$ 以串行的方式输出。在此基础上增加一个低电平异步复位信号 reset。

2）Python 交互模板

（1）加载比特流（带并行加载移位寄存器的 python 交互模板）。其代码如下：

```
from pynq import Overlay
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/.bit")
```

（2）GPIO 口命名的代码：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_CLK = AxiGPIO(gpio_0).channel1
gpio_reset = AxiGPIO(gpio_0).channel2
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_load = AxiGPIO(gpio_1).channel1
gpio_Sin = AxiGPIO(gpio_1).channel2
gpio_2 = overlay.ip_dict['axi_gpio_2']
gpio_D = AxiGPIO(gpio_2).channel1
gpio_5 = overlay.ip_dict['axi_gpio_5']
gpio_Q = AxiGPIO(gpio_5).channel1
gpio_Sout = AxiGPIO(gpio_5).channel2
```

(3) 验证移位寄存器串行到并行的功能,代码如下:

```
# 写GPIO需要和其位数相匹配的掩码，如gpio_Sin是1位的，那么它的掩码2进制数0b1
mask  = 0b1
# 验证串行到并行的转换

#复位
gpio_CLK.write(0b0,mask)
gpio_reset.write(0b0,mask)
gpio_load.write(0b0,mask)
gpio_CLK.write(0b1,mask)
# 第一个周期Sin=1
gpio_CLK.write(0b0,mask)
gpio_reset.write(0b1,mask)
gpio_Sin.write(0b1,mask)
gpio_CLK.write(0b1,mask)

# 第八个周期Sin=1
gpio_CLK.write(0b0,mask)
gpio_Sin.write(0b1,mask)
gpio_CLK.write(0b1,mask)


# 验证Q是否为0b110
Q=gpio_Q.read()
print(bin(Q))
```

(4) 验证移位寄存器并行转串行的功能,代码如下:

```
# 写GPIO需要和其位数相匹配的掩码，如gpio_Sin是1位的，那么它的掩码2进制数0b1
mask  = 0b1
mask1 = 0xff
# 验证并行到串行的转换
Sout=[]

#复位
gpio_CLK.write(0b0,mask)
gpio_reset.write(0b0,mask)
gpio_load.write(0b0,mask)
gpio_CLK.write(0b1,mask)
# 第一个周期
gpio_CLK.write(0b0,mask)
gpio_reset.write(0b1,mask)
gpio_load.write(0b1,mask)
gpio_D.write(0b10100111,mask1)
gpio_CLK.write(0b1,mask)
tmp=gpio_Sout.read()
Sout.append(tmp)

# 第二个周期
gpio_CLK.write(0b0,mask)
gpio_load.write(0b0,mask)
gpio_CLK.write(0b1,mask)
tmp=gpio_Sout.read()
Sout.append(tmp)
# 第八个周期
gpio_CLK.write(0b0,mask)
gpio_CLK.write(0b1,mask)
tmp=gpio_Sout.read()
Sout.append(tmp)
print(Sout)
```

3) 源代码

本实验源代码中所含文件介绍同8.2.2节。

4. 实验步骤

实验步骤同 8.2.2 节。

8.2.6 计数器

1. 实验目的

掌握基于 Verilog HDL 方式的计数器程序设计方法，通过 Vivado 软件和 PYNQ 平台进行开发调试，完成给出的四位计数器实验，并进一步设计 8 位计数器，并包含异步复位功能。

2. 实验要求

要求采用源代码中对应目录下的"基础模块"所提供的".v"源文件建立完整的工程，使用其中的测试文件进行整体仿真，仿真完成后生成 IP，并调用所生成的 IP 进行 I/O 的连接，构成完整的系统。最后生成流文件，并下板完成验证。

3. 实验说明

1) 实验原理

计数器主要是对脉冲的个数进行计数，N 位二进制计数器包含带有时钟和复位输入，N 位输出的时序算数电路。复位将输出初始化为 0，然后，计数器在每个时钟的上升沿递增 1，以二进制顺序输出所有 2^N 种可能值。可以参考源代码中对应目录下的"基础模块"所提供的".v"源文件，通过理解代码并完成 8 位包含异步复位功能计数器的设计。计数器电路符号如图 8-25 所示。

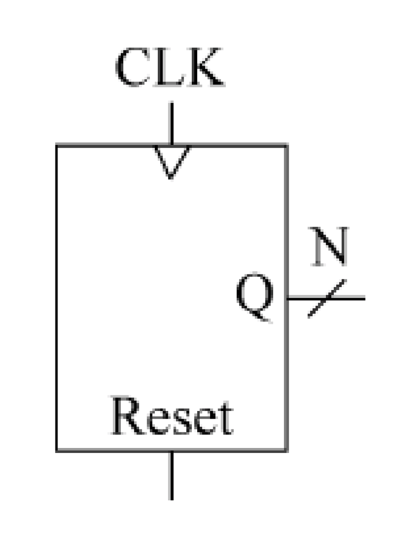


图 8-25 计数器电路符号

异步复位与同步复位的不同在于异步复位可以在复位输入为 0 时立刻进行复位而不用等到下一个时钟上升沿时才进行复位。

2) Python 交互模板

(1) 加载比特流的代码：

```
from pynq import Overlay                                                #调用Python库
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/......bit")          #bit文件路径
overlay?
```

(2) GPIO 口命名的代码：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_2 = overlay.ip_dict['axi_gpio_2']

clk = AxiGPIO(gpio_0).channel1                          #输入输出端口命名
rst_n = AxiGPIO(gpio_0).channel2
out0 = AxiGPIO(gpio_1).channel1
```

(3) 复位设置的代码：

```
import time                                             #调用时间函数
mask=0b1                                                #GPIO掩码
timedelay = 0.1                                         #设置时间延时
rst_n.write(0b0,mask)                                   #设置复位
rst_n.write(0b1,mask)
```

（4）通过循环显示结果的代码：

```
import time
N = 30
timedelay = 0.1
sizes = range(0,N,1)                    #设置循环次数
mask=0b1
for size in sizes:
    clk.write(0b0,mask)                 #创造时钟
    clk.write(0b1,mask)

    tmp=[]                              #创建数组保存结果

    a1 = out0.read()
    tmp.append(a1)

    print(tmp)                          #打印数据
```

3）源代码

本实验源代码中所含文件介绍同 8.2.2 节。

4. 实验步骤

实验步骤同 8.2.2 节。

8.2.7　有限状态机

1. 实验目的

学习三段式状态机的基本构成和工作原理，实现三段式状态机并对其加以改进和优化。

2. 实验要求

要求采用源代码中对应目录下的“基础模块”所提供的“.v”源文件建立完整的工程，使用其中的测试文件进行整体仿真，仿真完成后生成 IP，并调用所生成的 IP 进行 I/O 的连接，构成完整的系统。最后生成比特流文件，并下板完成验证。

3. 实验说明

1）实验原理

图 8-26(a)所示状态机含有三个状态分别是 S0、S1 和 S2。状态机默认处在 S0 状态，输出结果 0，如果 en=1 时，状态机进入状态 1，输出结果 1，否则状态不变；在状态 1 时，如果 en=1，状态机进入状态 2，输出结果 2，并且返回到 S0 状态。可以在完成上述实验后，再视情况完成图 8-26(b)的设计实验。

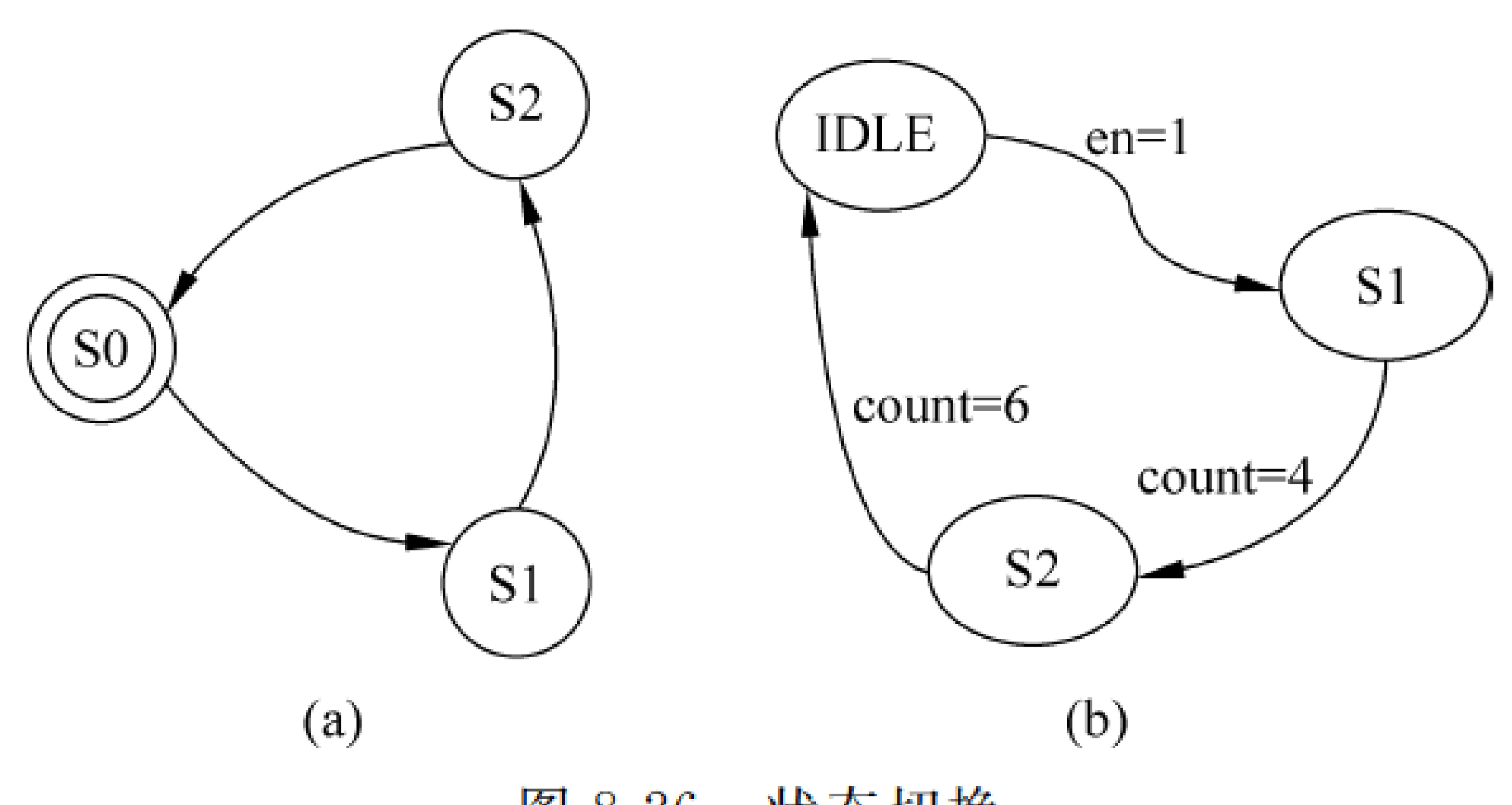


图 8-26　状态切换

2）Python 交互模板

（1）加载比特流的代码：

```
from pynq import Overlay                                          #调用Python库
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/......bit")     #bit文件路径
overlay?
```

（2）GPIO 口命名的代码：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_2 = overlay.ip_dict['axi_gpio_2']
gpio_3 = overlay.ip_dict['axi_gpio_3']
gpio_4 = overlay.ip_dict['axi_gpio_4']

clk = AxiGPIO(gpio_0).channel1                       #输入输出端口命名
rst_n = AxiGPIO(gpio_0).channel2
en = AxiGPIO(gpio_1).channel1
out0 = AxiGPIO(gpio_1).channel2
```

（3）复位设置的代码：

```
import time                                          #调用时间函数
mask=0b1                                             #GPIO掩码
timedelay = 0.1                                      #设置时间延时
rst_n.write(0b0,mask)                                #设置复位
rst_n.write(0b1,mask)
```

（4）通过循环显示结果的代码：

```
import time
N = 5
timedelay = 0.1
sizes = range(0,N,1)                                 #设置循环次数
mask = 0b1
en.write(0b1, mask)
tables = []
for size in sizes:
    clk.write(0b0,mask)                              #创造时钟
    clk.write(0b1,mask)

    tmp0=[]                                          #创建数组保存结果
    tmp0.append(size)

    a0 = out0.read()
    tmp0.append(a0)

    print(tmp0)                                      #打印数据
```

3）源代码

本实验源代码中所含文件介绍同 8.2.2 节。

4. 实验步骤

实验步骤同 8.2.2 节。

8.2.8 运算器/ALU

1. 实验目的

通过 Verilog HDL 方式实现 ALU 程序设计，通过 Vivado 软件和 PYNQ 平台进行开发调试，实现 ALU 的功能，通过更改调试文件来实现不同的运算功能，实验中所使用的 ALU 部分代码来自开源 RISC-V 处理器。

2. 实验要求

要求采用源代码中对应目录下的"基础模块"所提供的".v"源文件建立完整的工程，使用其中的测试文件进行整体仿真，仿真完成后生成 IP，并调用所生成的 IP 进行 I/O 的连接，构成完整的系统。最后生成比特流文件，并下板完成验证。

3. 实验说明

1）实验原理

ALU 的真值表如表 8-3 所示，普通 ALU 中包括表 8-3 给出的 12 种操作指令，每个 info 对应不同的指令类型，输入特定的 info 可以实现特定的指令操作，通过修改调试程序可以实现不同的计算功能。

表 8-3 ALU 真值表

指令类型	Info[31:0]	说　明
LUI	0xc008	Imm<<12(左移 12 位)
SUB	0x28	rs1－rs2
ADD	0x18	rs1＋rs2
ADDI	0x8018	rs1＋imm
SLT	0x1008	if(rs1<rs2) return 1;else return 0
SLTI	0x9008	if(rs1<imm) return 1;else return 0
XOR	0x48	rs1^rs2(异或)
OR	0x408	rs1|rs2(或)
AND	0x808	rs1&rs2(与)
SLL	0x88	rs1 逻辑左移 rs2 位
SRL	0x108	rs1 逻辑右移 rs2 位
SRA	0x208	rs1 算术右移 rs2 位

本实验与上节运算器实验不同之处为本实验通过 Verilog HDL 代码的方式将运算通路模块与普通 ALU 模块进行了连接，连接两个模块的文件称之为 ALU 顶层模块，通过顶层文件可以直接生成完整的 ALU 模块。

2）Python 交互模板

（1）加载比特流的代码：

```
from pynq import Overlay                                         #调用Python库
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/......bit")   #bit文件路径
overlay?
```

（2）GPIO 口命名的代码：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_rs1 = AxiGPIO(gpio_0).channel1                          #输入输出端口命名
gpio_rs2 = AxiGPIO(gpio_0).channel2
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_imm = AxiGPIO(gpio_1).channel1
gpio_info = AxiGPIO(gpio_1).channel2
gpio_2 = overlay.ip_dict['axi_gpio_2']
gpio_wdat = AxiGPIO(gpio_2).channel1
```

（3）GPIO 口读写的代码：

```
mask = 0xffffffff                                            #GPIO掩码
gpio_rs1.write(8,mask)                                       #写入数据
gpio_rs2.write(3,mask)
gpio_imm.write(0,mask)
gpio_info.write(0x18,mask)
wdat = gpio_wdat.read()
print(wdat)                                                  #打印数据
```

3）源代码

本实验源代码中所含文件介绍同 8.2.2 节。

4. 实验步骤

实验步骤同 8.2.2 节。

8.2.9 存储器

1. 实验目的

通过 Verilog HDL 方式实现存储器程序设计，使用 Vivado 软件和 PYNQ 平台进行开发调试，实现对基本存储器模块的使用，利用多个存储器模块搭建更大的存储器系统。

2. 实验要求

要求采用源代码中对应目录下的“基础模块”所提供的“.v”源文件建立完整的工程，使用其中的测试文件进行整体仿真，仿真完成后生成 IP，并调用所生成的 IP 进行 I/O 的连接，构成完整的系统。最后生成流文件，并下板完成验证。

3. 实验说明

1）实验原理

基础模块中包含两个独立的存储器单元，分别为 mem2 和 mem2_1，写入和读出数据时都需要选择针对哪个存储器进行操作，通过片选信号可以控制要写入和读出的为哪一个存储器，片选信号为 info 的第 0 和 1 位，当使用 mem2 时，info 最低的两位要使用数值 01 才可以正常地写入和读出；当使用 mem2_1 时，info 最低的两位为 10，如果片选信号错误存储器将不能正常地写入和读出数据。

本实验与上节存储器实验不同之处为本实验通过 Verilog HDL 代码的方式将两个存储器进行连接，将两个存储器模块组合成一个更大的存储器模块，连接两个模块的文件称为顶层模块。

完成上述实验后可以结合所学情况练习增加一个片选信号为 11 的存储器到整体的存储器模块中，并能执行正确的数据写入写出。

2）Python 交互模板

（1）加载比特流的代码：

```
from pynq import Overlay                                              #调用Python库
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/......bit")        #bit文件路径
overlay?
```

（2）GPIO 口命名的代码：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_2 = overlay.ip_dict['axi_gpio_2']

info = AxiGPIO(gpio_0).channel1                   #输入输出端口命名
wdata = AxiGPIO(gpio_0).channel2
rdata1 = AxiGPIO(gpio_1).channel1
rdata = AxiGPIO(gpio_1).channel2
```

（3）GPIO 口读写的代码：

```
mask = 0xffffffff                                  #GPIO掩码
info.write(0x15, mask)                             #输入信息
wdata.write(0x99999999, mask)                      #输入数据
info.write(0x11, mask)
out1 = hex(rdata1.read())
print('out1 =', out1)                              #打印数据

info.write(0x26, mask)
wdata.write(0x88880000, mask)
info.write(0x22, mask)
out2 = hex(rdata.read())
print('out2 =', out2)
```

3）源代码

本实验源代码中所含文件介绍同 8.2.2 节。

4. 实验步骤

实验步骤同 8.2.2 节。

8.2.10　寄存器堆

1. 实验目的

通过 Verilog HDL 方式实现寄存器程序设计，使用 Vivado 软件和 PYNQ 平台进行开发调试，通过对基本寄存器模块的使用，能够在单个寄存器基础上搭建寄存器组，实现具有 5 个寄存器的数据通路。

2. 实验要求

要求采用源代码中对应目录下的“基础模块”所提供的“.v”源文件建立完整的工程，使用其中的测试文件进行整体仿真，仿真完成后生成 IP，并调用所生成的 IP 进行 I/O 的连接，构成完整的系统。最后生成流文件，并下板完成验证。

3. 实验说明

1）实验原理

如图 8-27 所示，数据写入时默认暂存到寄存器 DR 中，DR 的输出可以更新 R0～R4 中

的任意一个寄存器，具体更新到哪一个寄存器需要由各个寄存器的使能信号决定，当某个寄存器使能为 1 时，寄存器 DR 中的数据进入相应的寄存器中，端口 A 和 B 都具有选择功能，通过输入的选择信号，它们可以选择任一个寄存器的数据进行输出。

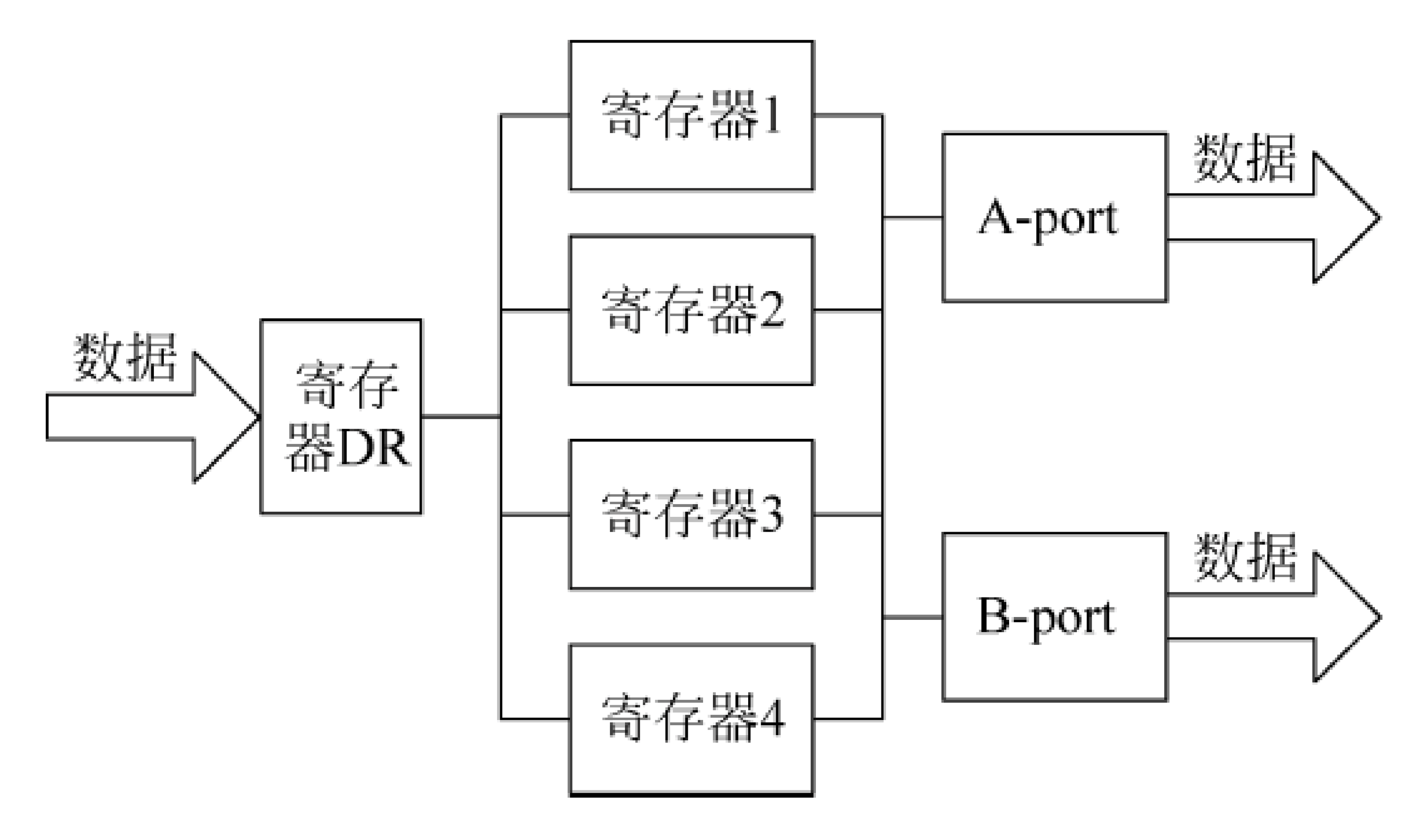


图 8-27　寄存器组示意图

基础模块中给出了 DR、R0、R1 三个寄存器模块组合的代码，实验时可以先行验证这三个模块的运行结果，再结合学习情况扩充成 5 个寄存器的数据通路。

2）Python 交互模板

(1) 加载比特流的代码：

```
from pynq import Overlay                                          #调用Python库
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/......bit")     #bit文件路径
overlay?
```

(2) GPIO 口命名的代码：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_2 = overlay.ip_dict['axi_gpio_2']
gpio_3 = overlay.ip_dict['axi_gpio_3']

a_sel = AxiGPIO(gpio_0).channel1                  #输入输出端口命名
b_sel = AxiGPIO(gpio_0).channel2
DR_en = AxiGPIO(gpio_1).channel1
DR_in = AxiGPIO(gpio_1).channel2
R0_en = AxiGPIO(gpio_2).channel1
R1_en = AxiGPIO(gpio_2).channel2
a_port = AxiGPIO(gpio_3).channel1
b_port = AxiGPIO(gpio_3).channel2
```

(3) GPIO 口读写的代码：

```
mask = 0xffff                                     #GPIO掩码
DR_en.write(1,mask)                               #写入数据
DR_in.write(0xffff,mask)
R0_en.write(1,mask)
a_port.write(0b000,mask)
data = hex(a_port.read())                         #转换成16位数据
print(data)                                       #打印数据
```

3）源代码

本实验源代码中所含文件介绍同 8.2.2 节。

4. 实验步骤

实验步骤同 8.2.2 节。

8.2.11 总线

1. 实验目的

了解总线的基本构成和工作原理,通过 Verilog HDL 方式实现总线程序设计,使用 Vivado 软件和 PYNQ 平台进行开发调试,实现通过总线模块进行的数据传输。

2. 实验要求

要求采用源代码中对应目录下的"基础模块"所提供的".v"源文件建立完整的工程,使用其中的测试文件进行整体仿真,仿真完成后生成 IP,并调用所生成的 IP 进行 I/O 的连接,构成完整的系统。最后生成比特流文件,并下板完成验证。

3. 实验说明

1) 实验原理

实现基于总线进行数据传输的寄存器数据写入与读取。本实验采用蜂鸟 E203 处理器中自定义总线协议 ICB(Internal Chip Bus),该总线应用于蜂鸟 E203 处理器中,其结合 AXI 总线和 AHB 总线的优点,兼具高速性和易用性。其总线通道结构如图 8-28 所示。

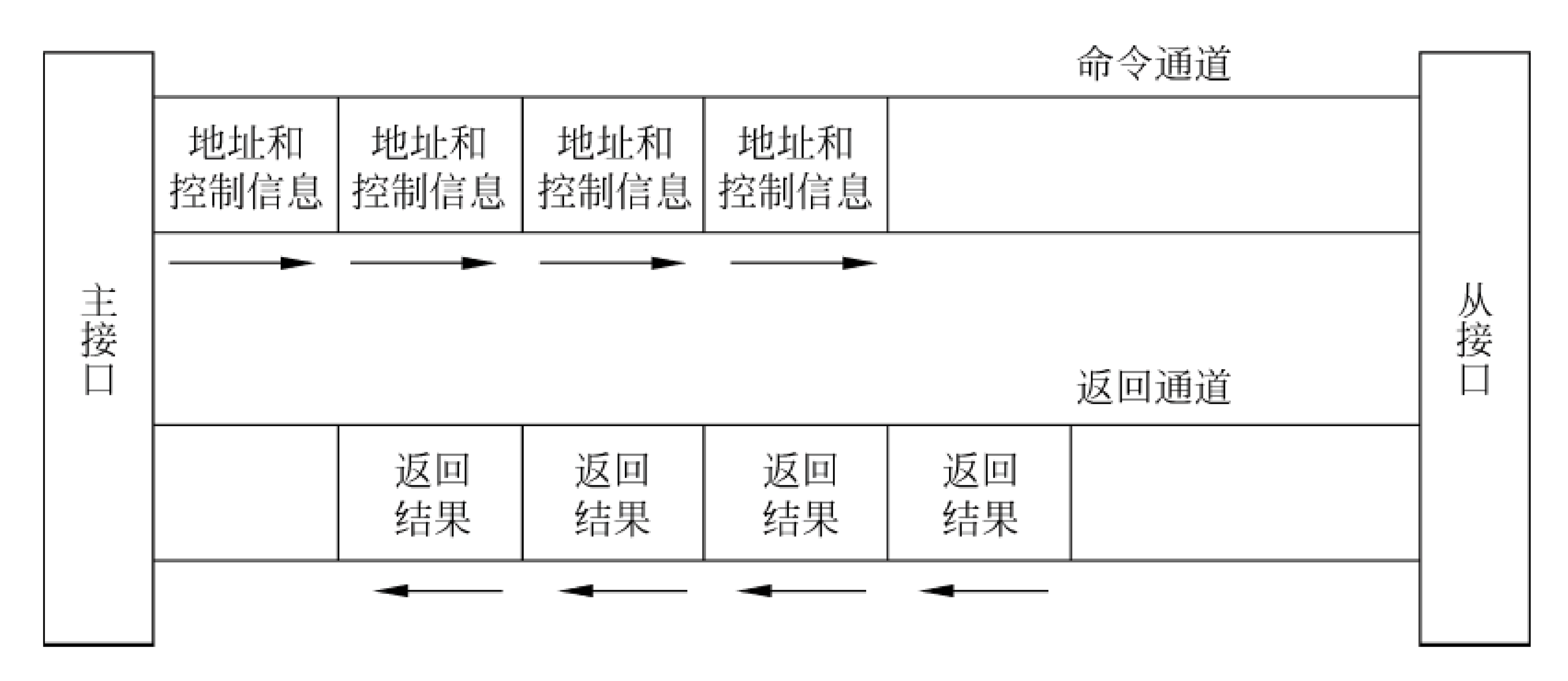


图 8-28 总线通道结构

ICB 总线信号列表如表 8-4 所示:命令通道(Command Channel)主要用于主设备向从设备发起读写请求。返回通道(Response Channel)主要用于从设备向主设备返回读写结果。

表 8-4 ICB 总线信号

通道	方向	宽度	信号名	介 绍
命令通道	Output	1	Icb_cmd_valid	主设备向从设备发送读写请求信号
	Input	1	Icb_cmd_ready	从设备向主设备返回读写接收信号
	Output	8	Icb_cmd_addr	读写地址
	Output	1	Icb_cmd_read	读写指示
	Output	16	Icb_cmd_wdata	写操作的数据
	Output	16	Icb_cmd_wmask	写操作时的字节掩码
返回通道	Input	1	Icb_rsp_valid	从设备向主设备发送读写反馈请求信号
	Output	1	Icb_rsp_ready	主设备向从设备返回读写反馈接收信号
	Input	16	Icb_rsp_rdata	读反馈的数据
	Input	1	Icb_rsp_err	读或写反馈的错误标志

ICB总线时序有若干种，图8-29为典型时序中的其中一种：主设备向从设备通过ICB的Command Channel发送写操作请求(icb_cmd_read为低)，从设备立即接收该请求(icb_cmd_ready为高)。从设备在同一个周期返回读结果且结果正确(icb_rsp_err为低)，主设备立即接收该结果(icb_rsp_ready为高)。

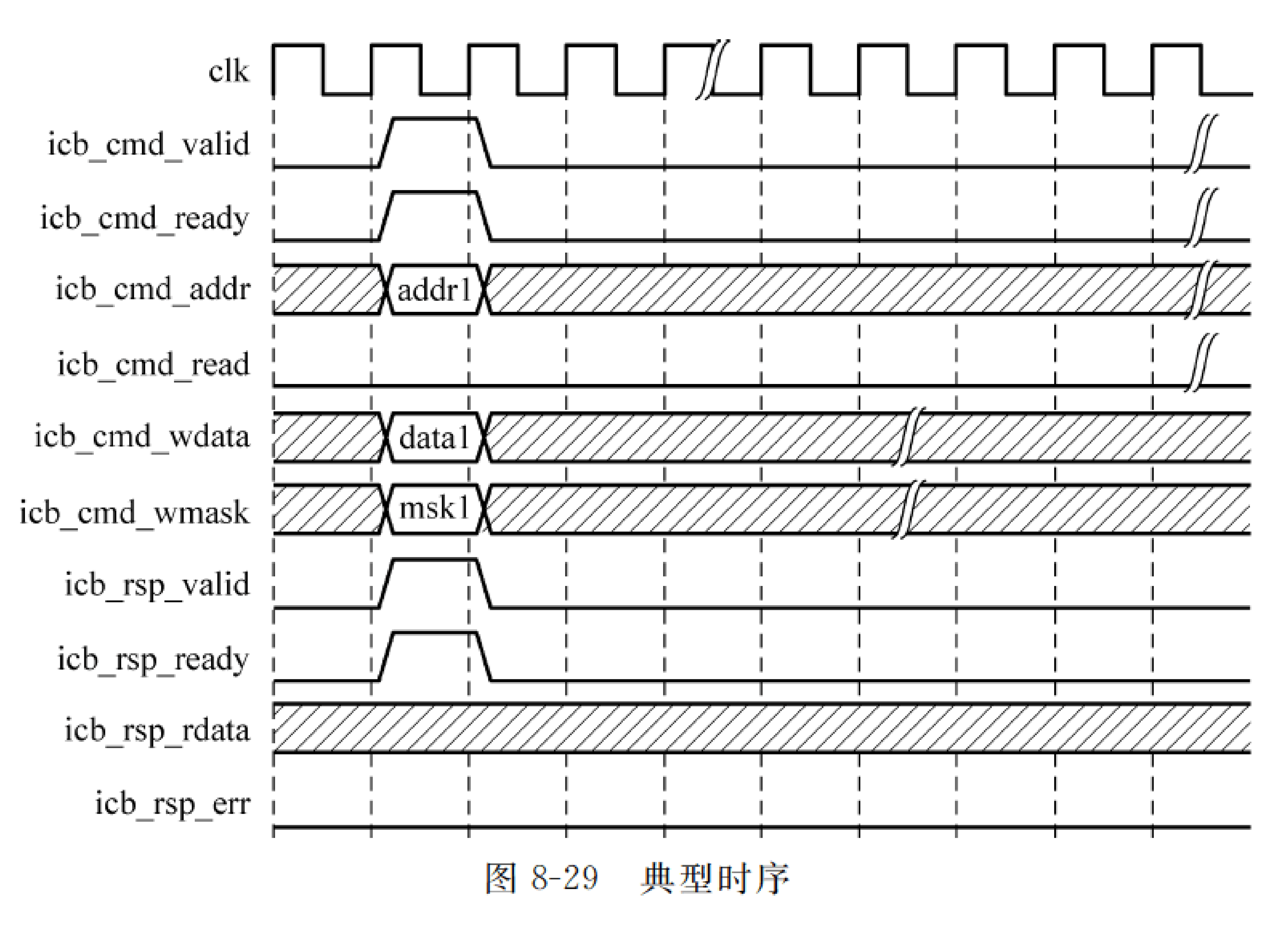


图 8-29 典型时序

源代码中包含总线模块和ram模块两个部分，其中PYNQ的PS端为主模块，ram模块为从模块，用户通过PS端发送数据，总线模块接收数据并将数据传入ram模块中，完成本实验后可以根据学习情况和ICB总线协议自行设计一个从模块并进行验证。

2) Python交互模板

(1) 加载比特流的代码：

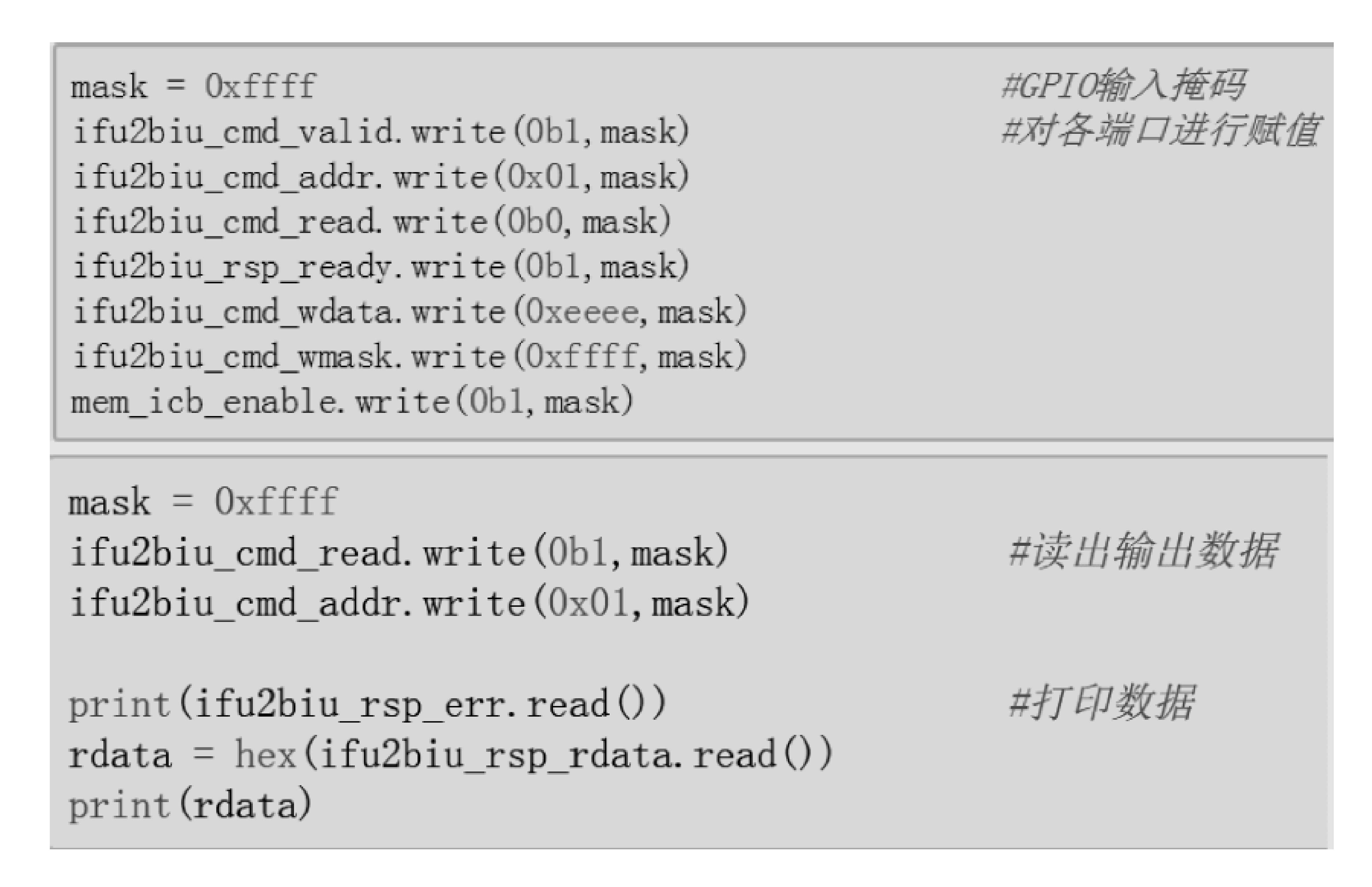
```
from pynq import Overlay                                          #调用Python库
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/......bit")     #bit文件路径
overlay?
```

(2) GPIO口命名的代码：

```
gpio_0 = overlay.ip_dict['axi_gpio_0']
gpio_1 = overlay.ip_dict['axi_gpio_1']
gpio_2 = overlay.ip_dict['axi_gpio_2']
gpio_3 = overlay.ip_dict['axi_gpio_3']
gpio_4 = overlay.ip_dict['axi_gpio_4']
gpio_5 = overlay.ip_dict['axi_gpio_5']

ifu2biu_cmd_valid = AxiGPIO(gpio_0).channel1          #对端口进行命名
ifu2biu_cmd_addr = AxiGPIO(gpio_0).channel2
ifu2biu_cmd_read = AxiGPIO(gpio_1).channel1
ifu2biu_cmd_wdata = AxiGPIO(gpio_1).channel2
ifu2biu_cmd_wmask = AxiGPIO(gpio_2).channel1
ifu2biu_rsp_ready = AxiGPIO(gpio_2).channel2
mem_icb_enable = AxiGPIO(gpio_3).channel1
icb2biu_cmd_ready = AxiGPIO(gpio_3).channel2
ifu2biu_rsp_valid = AxiGPIO(gpio_4).channel1
ifu2biu_rsp_err = AxiGPIO(gpio_4).channel2
ifu2biu_rsp_rdata = AxiGPIO(gpio_5).channel1
```

（3）GPIO 口读写的代码：



```
mask = 0xffff                              #GPIO输入掩码
ifu2biu_cmd_valid.write(0b1,mask)          #对各端口进行赋值
ifu2biu_cmd_addr.write(0x01,mask)
ifu2biu_cmd_read.write(0b0,mask)
ifu2biu_rsp_ready.write(0b1,mask)
ifu2biu_cmd_wdata.write(0xeeee,mask)
ifu2biu_cmd_wmask.write(0xffff,mask)
mem_icb_enable.write(0b1,mask)
```

```
mask = 0xffff
ifu2biu_cmd_read.write(0b1,mask)           #读出输出数据
ifu2biu_cmd_addr.write(0x01,mask)

print(ifu2biu_rsp_err.read())              #打印数据
rdata = hex(ifu2biu_rsp_rdata.read())
print(rdata)
```

3）源代码

本实验源代码中所含文件介绍同 8.2.2 节。

4. 实验步骤

实验步骤同 8.2.2 节。

8.2.12 微程序控制器

1. 实验目的

掌握基于 PYNQ 平台及 Vivado 工具的时序逻辑电路的开发流程及调试方式。本实验提供了一个简单的 CPU 模型机，要求学生阅读其 Verilog HDL，理解其工作原理，并补充完善微程序控制器中微程序的设计，使其可以仿真并下板运行。

2. 实验要求

要求采用源代码中对应目录下的“基础模块”所提供的“.v”源文件建立完整的工程，使用其中的测试文件进行整体仿真，仿真完成后生成 IP，并调用所生成的 IP 进行 I/O 的连接，构成完整的系统。最后生成比特流文件，并下板完成验证。

3. 实验说明

1）实验原理

（1）CPU 数据通路：如图 8-30 所示，每一条机器指令的执行过程都可分为两个阶段：取指阶段和执行阶段。

程序计数器(PC)是一条指令的起点，所以在取指阶段，首先由 PC 输出将要执行的机器指令的地址，接着指令存储器(Imem)将与该地址对应的机器指令输出到指令寄存器 IR，同时 PC 指向下一条要执行的指令，为下次取指做准备。

接着，IR 将指令的操作码送入指令译码器(Decoder)，译码器中存储有与之对应的微程序的入口地址，将其送入到微程序计数器(MPC)，最终由 MPC 确定将要执行的微指令地址，将该条微指令从微指令控制存储器(CUmem)中读出，将这些控制信号发送送到 CPU 的各个组件，指令便开始执行。

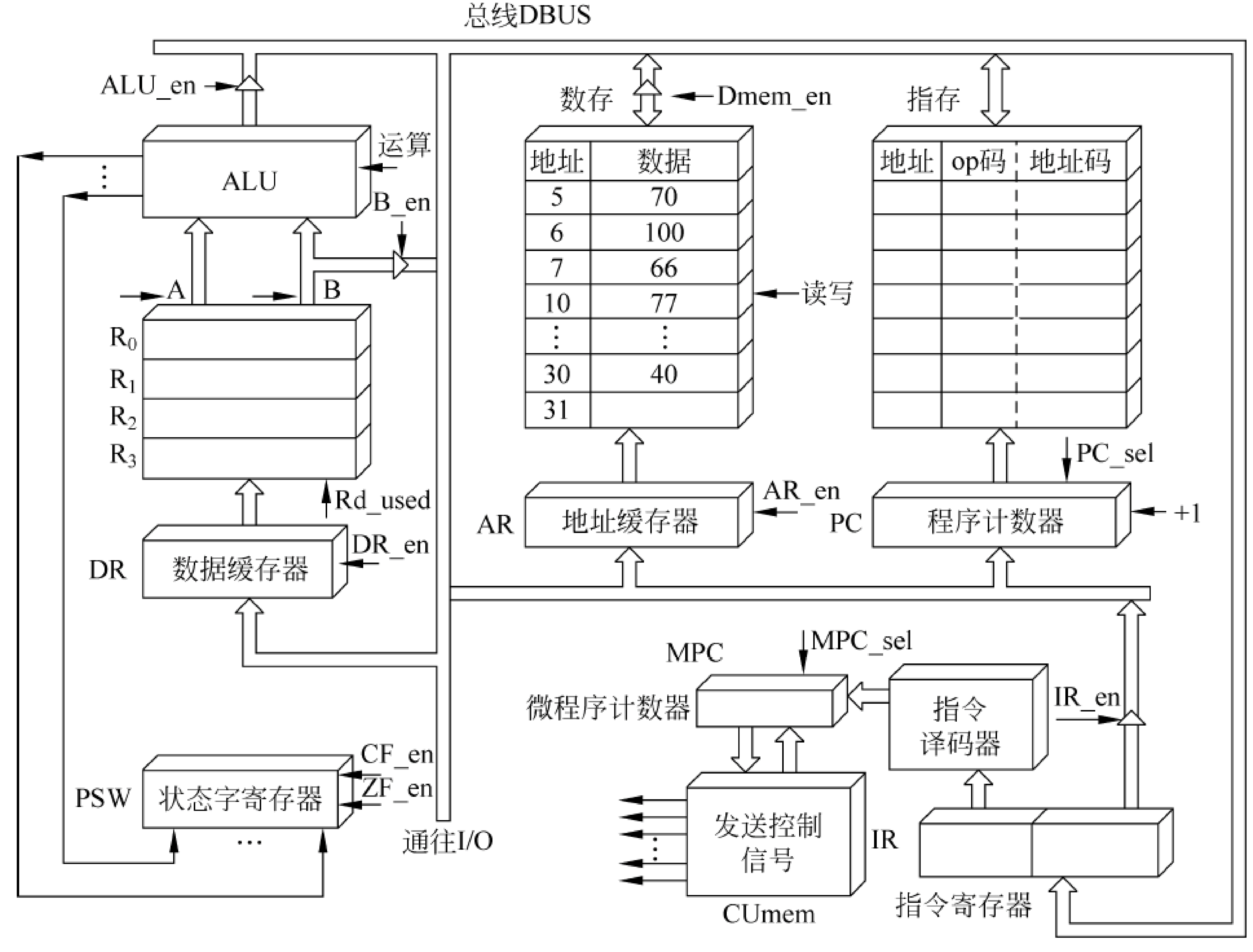


图 8-30　CPU 数据通路图

(2) 指令设计：该模型机设计了 3 条数据转移类指令(MOV,LAD,STO),7 条运算类指令(ADD,SUB,INC,DEC,AND,OR,NOT),7 条跳转类指令(JEQ,JNE,JHI,JHS,JLO,JLS,JMP)以及比较指令(CMP)和停机指令(HLT),共 19 条指令。其中,仅有 LAD 与 STO 可以访问存储器,大多数的指令采用寄存器寻址,LAD 和 STO 的寻址方式为间接寻址,跳转指令及部分第二操作数为立即数的指令采用立即寻址方式。

具体指令设计见源代码资料包中本实验文件夹中的“附录 2”。

(3) 取指微指令的实现：

第一条微指令如图 8-31 所示。

ZF_en	PC_en	PC_sel	Imem_en	MPC_en	MPC_sel	cs	we
0	0	0	1	1	10	0	0
ALU_en	B_en	Dmem_en	IR_data_en	CF_en	Rd_used	RsA_used	RsB_used
0	0	0	0	0	0	0	0
AR_en	outenab	M	S(4 位)		Next_addr(16 位)		
0	0	0	0000		0000 0000 0000 0001		

图 8-31　第一条微指令

① 第一个周期：PC 中的地址送入指令寄存器 Imem,通过发送控制信号 Imem_en,使该地址对应的指令操作码(Opcode)从 Imem 经过总线 DBUS 进入指令译码器,经过翻译,从指令译码器流出的数据变成了这条指令在 CUmem 中的入口地址,入口地址被输入到微程序计数器 MPC 的 MPC_in 中。同时发送了控制信号 MPC_sel=“10”,代表着下一条微指令地址将从 CU_mem 传回的 next_addr 中获取(此处 next_addr 即为取指程序的第二条微指令地址)。

第二条微指令如图 8-32 所示。

ZF_en	PC_en	PC_sel	Imem_en	MPC_en	MPC_sel	cs	we
0	1	0	0	1	11	0	0
ALU_en	B_en	Dmem_en	IR_data_en	CF_en	Rd_used	RsA_used	RsB_used
0	0	0	0	0	0	0	0
AR_en	outenab	M	S(4 位)		Next_addr(16 位)		
0	0	0	0000		01 0000 0000 0000		

图 8-32　第二条微指令

② 第二个周期：由于控制单元发送的微程序计数器 MPC 的选择信号 MPC_sel＝“11”（意为将 MPC_in 输出），指令的入口地址便被送入 CUmem，CUmem 便从该指令入口地址开始，逐条发送该指令对应的控制信号。同时，可以注意到表中的第二行，PC_en 被置为“1”，且 PC_sel＝“00”（意为 PC＝PC＋1），从而使 PC 指向下一条指令，为下一次取指做好准备。

（4）代码层次结构：

```
|-- tinycpu.v
    |-- ALU.v
    |-- DBUS.v
    |-- Regsfile.v
    |-- Dmem.v
    |-- Imem.v
    |-- Program_Counter.v
    |-- Decoder.v
    |-- MPC.v
    |-- CUmem.v
    |-- psw.v
```

具体代码请阅读基础模块文件夹。

2）Python 交互模板

（1）加载比特流的代码：

```
#==============================================================
#                  Load overlay
#==============================================================
from pynq import Overlay
from pynq.lib import AxiGPIO
overlay = Overlay("/home/xilinx/jupyter_notebooks/wzg/cpu/design_3.bit")
```

（2）GPIO 口的命名代码：

```
#==============================================================
#                GPIO口的命名
#==============================================================
gpio_5 = overlay.ip_dict['axi_gpio_5']
gpio_m0 = AxiGPIO(gpio_5).channel1
gpio_m1 = AxiGPIO(gpio_5).channel2
gpio_6 = overlay.ip_dict['axi_gpio_6']
gpio_m2 = AxiGPIO(gpio_6).channel1
gpio_m3 = AxiGPIO(gpio_6).channel2
gpio_7 = overlay.ip_dict['axi_gpio_7']
gpio_m4 = AxiGPIO(gpio_7).channel1
gpio_m5 = AxiGPIO(gpio_7).channel2
```

(3) 验证 CPU 是否成功排序,其代码如下:

```
#==========================================================
#                  GPIO口的读写
#==========================================================
import time
timedelay = 0.0001
time.sleep(timedelay )
tmp=[]
m0 = gpio_m0.read()
tmp.append(m0)
m1 = gpio_m1.read()
tmp.append(m1)
m2 = gpio_m2.read()
tmp.append(m2)
m3 = gpio_m3.read()
tmp.append(m3)
m4 = gpio_m4.read()
tmp.append(m4)
m5 = gpio_m5.read()
tmp.append(m5)
print(tmp)

[0, 1, 4, 5, 6, 10]
```

3) 源代码

本实验源代码中所含文件介绍同 8.2.1 节。

4. 实验步骤

(1) 将 CUmem 中的微指令设计完整,将"基础模块"中的所有.v 源文件添加到工程中。

(2) 使用本实验对应源代码中的 simu_cpu 仿真文件进行仿真,观察 m0～m5 的值是否为排序后的值(0,1,4,5,6,10)。因为仿真时间超过默认时间,所以需要在 simulation setting 中将 simulation runtime 设置为 40μs。

(3) 将仿真后的设计生成原理图添加到 IPrepository。

(4) 按照图 8-33 所示和接口模块相连接。

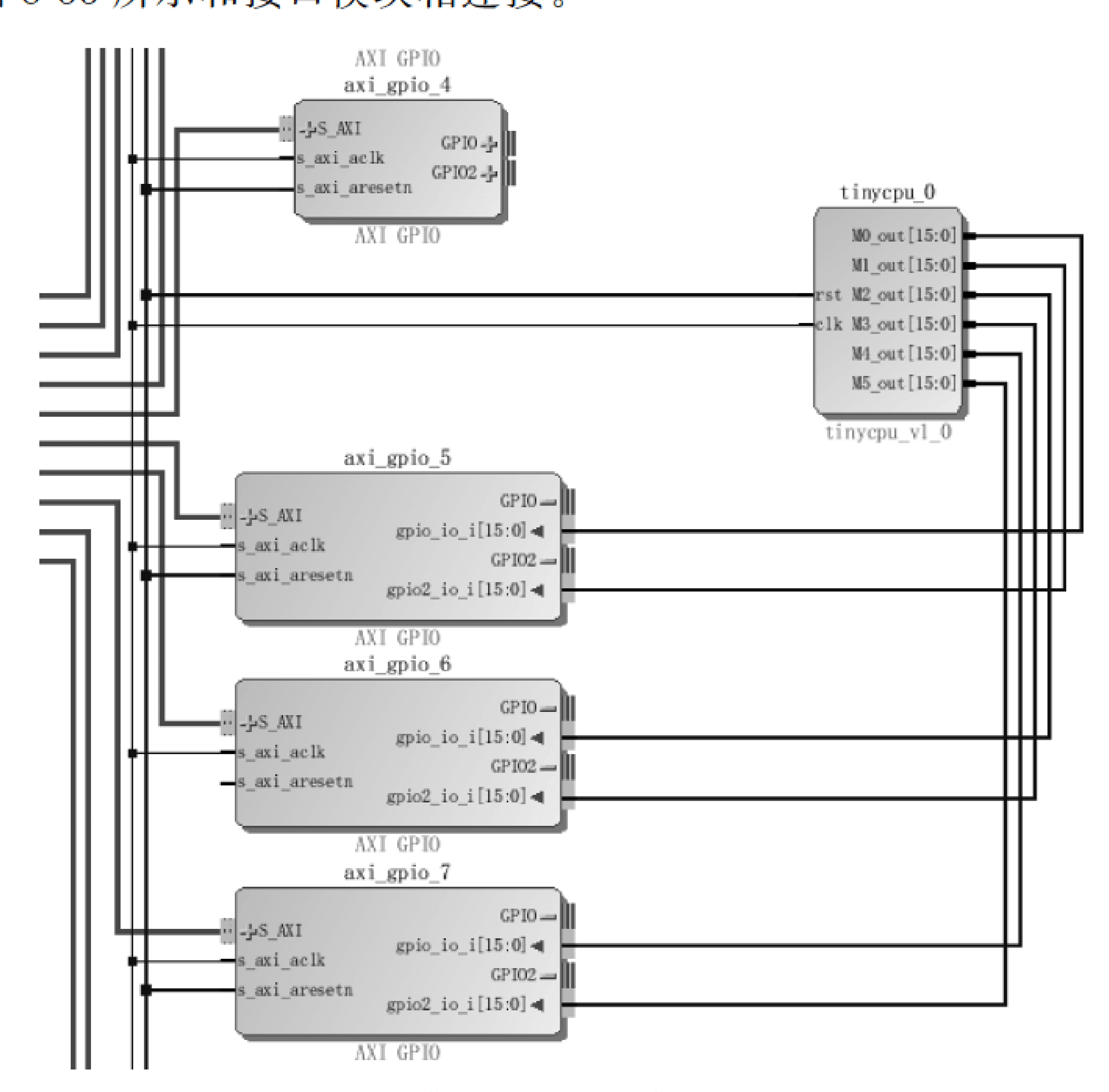


图 8-33 模型机与接口模块连接方式

(5) 重新综合、实现及生成比特流文件，并将比特流文件及对应的 tcl 文件上传到所用的 PYNQ 节点。

(6) 下板实验，使用本实验对应材料中的 Python 调试文件，观察真实运行后的 CPU 模型机是否正常工作。

8.2.13 中断

1. 实验目的

通过开源 RISC_V 处理器(E203)这一实例，掌握理解 RISC_V 架构的中断异常处理机制。

2. 实验要求

在著名开源网站 GitHub 上搜索 e200_opensource，下载该项目的源代码(或查看源代码资料包中 8.2.15 文件夹中的相关资料)，文件目录中的 rtl 文件夹包含了实验中要使用的全部源代码，应用图 8-34 中的 core 和 general 文件夹中的“.v”文件建立工程，根据给出的仿真文件和自测试文件分析仿真波形图，理解中断的处理机制。



图 8-34 文件目录

3. 实验说明

1) 实验说明

中断和异常都可以打断程序的正常执行，且它们之间最大的区别是起因内外有别。因此，中断和狭义上的异常可以看作是一种广义上的异常。一般异常可分为同步异常、精确异步异常和非精确异步异常。

RISC_V 架构虽然支持多种工作模式，但是要求机器模式是必须具备的模式。因此，在这仅简单介绍基于机器模式的异常处理机制。

(1) RISC_V 架构中与异常相关的 CSR(控制状态)寄存器如下：

① mtvec：定义进入异常服务程序的 PC 地址；

② mcause：反映进入异常的原因；

③ mtval：反映进入异常的信息；

④ mepc：用于保存异常的返回地址；

⑤ mstatus：用于反映中断全局使能；

⑥ mie：用于控制不同类型的局部使能；

⑦ mip：反映不同类型中断的等待状态。

(2) 当处理器进入异常时，处理步骤如下：

① 处理器停止当前程序流，转而从 CSR 寄存器 mtvec 定义的 PC 地址开始执行异常服务程序。

② 硬件同时更新 mcause、mepc、mtval 和 mstatus 四个寄存器。

在执行异常服务程序时会通过 mcause 中异常的编号来决定进一步跳转到更具体的异常服务程序。

(3) 当程序完成异常处理后,需要从异常服务程序中退出,并返回主程序。RISC_V 架构定义了一个专门的退出异常指令 MRET。处理器执行 MRET 指令后会停止执行当前程序流,同时更新 mstatus 寄存器。

有关蜂鸟 E203 处理器的更详细的中断机制请参考《手把手教你设计 CPU——RISC-V 处理器》进行学习。

2) 源代码

本书配套源代码中,为此实验提供了如下材料:

(1) "test. v":是对用户整体工程进行仿真时所需的 Testbench 示例程序。

(2) "rv32ui_p_addi. data":是用于初始化 ITCM 的 addi 指令测试程序文件。

(3) "rv32ui_p_addi. dump":是上述测试程序的反汇编文件。

4. 实验步骤

(1) 依据实验要求下载需要的源代码并完成工程的建立,再将 rv32ui_p_addi. data 文件添加到工程中。

(2) 对工程进行仿真,对照反汇编文件分析波形图,如图 8-35 所示。

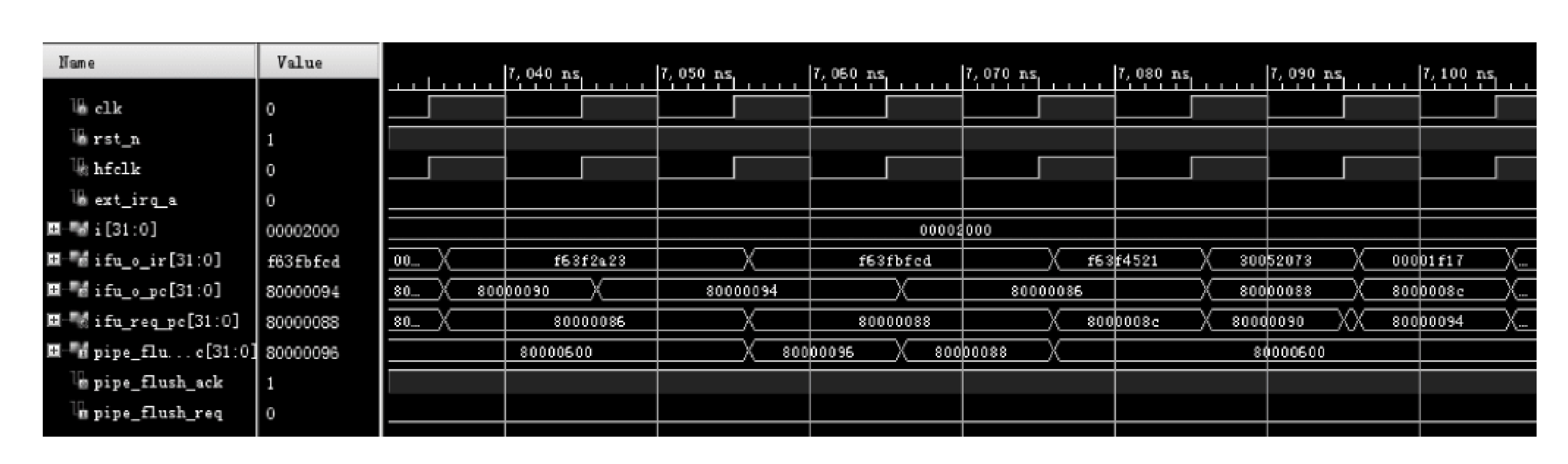


图 8-35 仿真波形图

8.2.14 基于开源项目的 CPU 内核的实现

1. 实验目的

了解开源处理器的相关知识,阅读开源的蜂鸟 E200 系列处理器的相关资料和 Verilog 代码,深入了解此款处理器内核部分的代码。

2. 实验要求

在著名开源网站 GitHub 上搜索 e200_opensource,下载该项目的源代码,文件目录中的 rtl 文件夹包含了实验中要使用的全部源代码,从代码中抽出内核部分代码,并能保证其能进行正确的仿真或实现。

3. 实验说明

蜂鸟 E200 处理器不仅完成了内核的实现,还搭配完整的 SoC 平台,其结构如图 8-36 所示,本实验要求读者从 E200 完整的代码中抽出内核的相关代码。

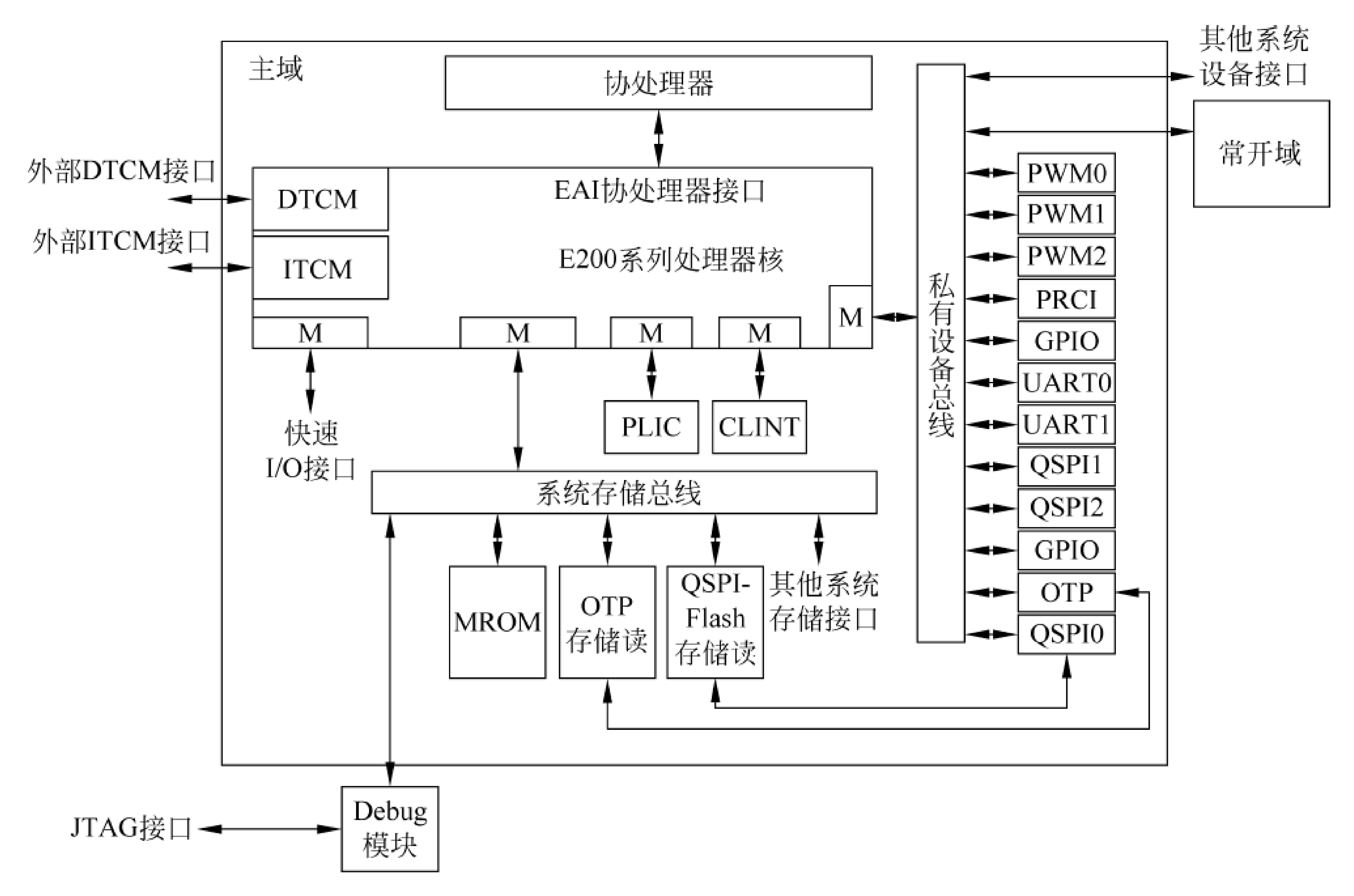


图 8-36　E200 系列处理器配套 SoC

4. 实验步骤

(1) 查阅有关蜂鸟 E200 系列处理器的相关资料,阅读其源代码并找出内核的相关代码。

(2) 对内核相关代码进行仿真,观察仿真结果是否满足设计要求。

(3) 将本实验对应.v 文件生成 IP 核。

(4) 调用 IP 核,进行 I/O 处理,将系统的逻辑 I/O 与 PS 相连接,以便进行调试。

(5) 重新综合、实现及生成比特流文件,并将比特流文件及对应的 tcl 文件上传到所用 PYNQ 节点。

(6) 下板实验,自行设计 Python 调试文件,观察真实运行后的系统是否正常工作。

8.2.15 为开源 CPU 增加指令

1. 实验目的

了解开源处理器的相关知识,阅读开源的蜂鸟 E200 系列处理器的相关资料和 Verilog 代码,了解处理器内核部分的完整工作机制和流程。

2. 实验要求

本实验提供了一个不完整的蜂鸟 E200 系列处理器项目,通过补全代码使 E203 处理器能正常运行,通过仿真方式对处理器进行功能验证。

3. 实验说明

处理器中缺少的部分都位于 E200 系列处理器的流水线第 2 级——执行阶段,该阶段缺少部分指令的译码和执行代码,读者通过了解内核部分的工作机制和工作流程,将蜂鸟 E200 系列内核中缺少的部分补充完整,使其正常运行。

4. 实验步骤

(1) 查阅有关蜂鸟 E200 系列处理器的相关资料，查看源代码，了解处理器内核部分的的完整工作机制和流程。

(2) 对内核相关代码进行仿真，观察仿真结果是否满足设计要求。

(3) 将本实验对应.v 文件生成 IP 核。

(4) 调用 IP 核，进行 I/O 处理，将系统的逻辑 I/O 与 PS 相连接，以便进行调试。

(5) 重新综合、实现及生成比特流文件，并将比特流文件及对应的 tcl 文件上传到所用 PYNQ 节点。

(6) 下板实验，自行设计 Python 调试文件，观察真实运行后的系统是否正常工作。

8.2.16 增加开源 CPU 的流水线级数

1. 实验目的

了解处理器的流水线基本概念，通过开源的蜂鸟 E203 处理器的 Verilog 代码深入理解此款处理器的流水线工作流程，并对此款处理器进行改进，将此款开源处理器的 2 级流水线改为 3 级流水线。

2. 实验要求

在著名开源网站 GitHub 上搜索 e200_opensource，并下载该项目的源代码，文件目录中的 rtl 文件夹包含了实验中要使用的全部源代码，应用图 8-37 中的 core 和 general 文件夹中的“.v”文件建立工程，通过对工程文件的修改完成流水线的增加，通过仿真观察流水线改变后的波形图结果，仿真完成后生成 IP，并调用所生成的 IP 进行 I/O 的连接，构成完整的系统。最后生成比特流文件，并下板完成验证。



图 8-37 文件目录

3. 实验说明

本实验中用到的蜂鸟 E203 处理器具有两级流水线，其结构如图 8-36 所示，其要点如下：

(1) 流水线的第 1 级为“取指(由 IFU 完成)”。

(2) 蜂鸟 E203 的“译码”“执行”“写回”均处于同一时钟周期，属于流水线第 2 级。

增加流水线的长度可以通过在蜂鸟 E203 处理器第 2 级流水中增加流水线寄存器来实现，关于流水线方面的问题可以参考网上资料或者查阅相关书籍进一步了解，有关蜂鸟 E203 处理器的更多内容也可以参考《手把手教你设计 CPU——RISC-V 处理器》进行学习。

4. 实验步骤

(1) 查阅有关处理器流水线的相关资料，理解本实验的相关代码，对 E203 处理器的源代码进行修改实现 3 级流水。

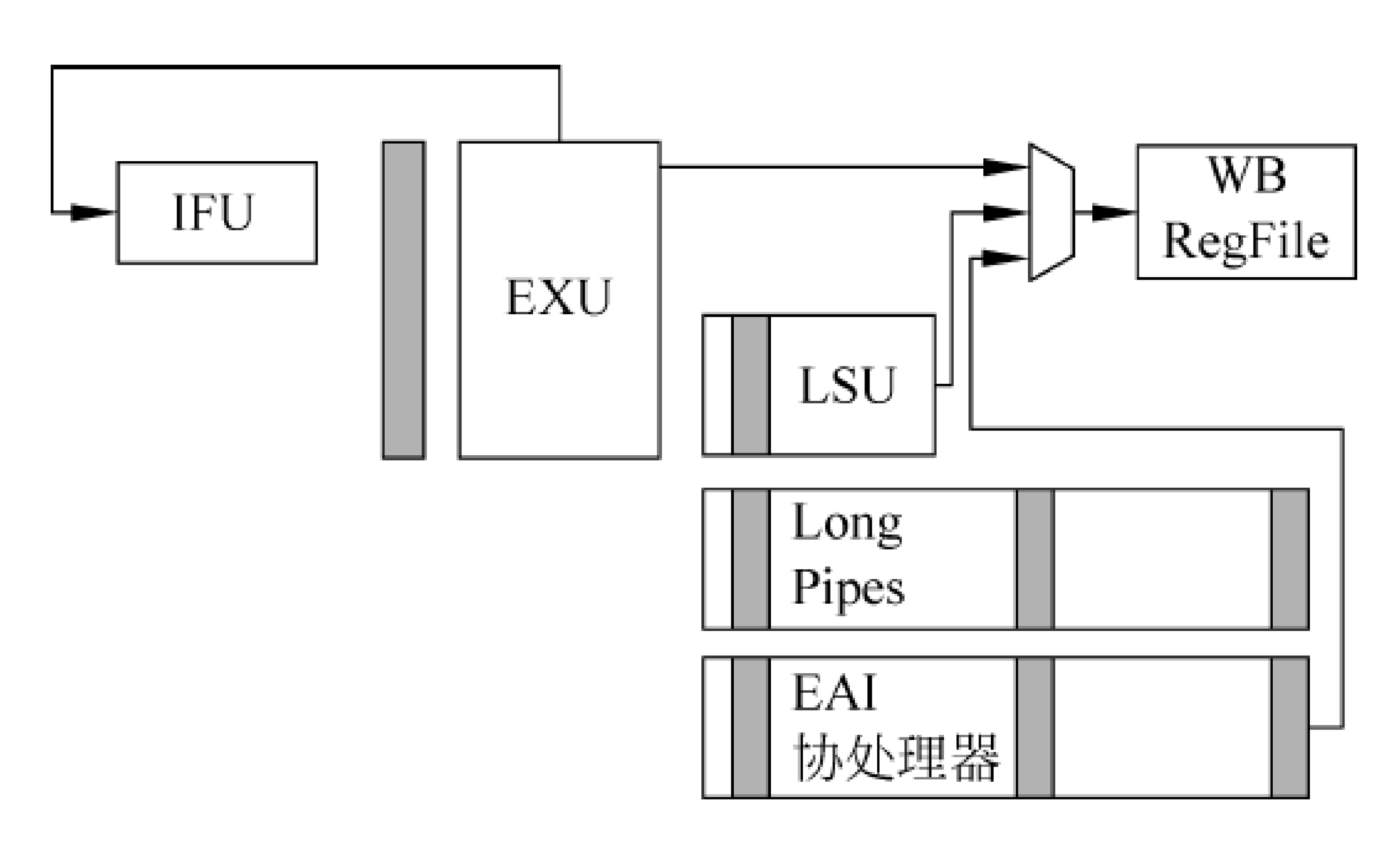


图 8-38 E203 流水线结构

（2）通过对处理器进行仿真，观察仿真结果是否实现 3 级流水结构。

（3）将本实验对应.v 文件生成 IP 核。

（4）调用 IP 核，进行 I/O 处理，将系统的逻辑 I/O 与 PS 相连接，以便进行调试。

（5）重新综合、实现及生成比特流文件，并将比特流文件及对应的 tcl 文件上传到所用 PYNQ 节点。

（6）下板实验，自行设计 Python 调试文件，观察真实运行后的系统是否正常工作。

参 考 文 献

[1] 白中英. 计算机组成原理[M]. 5 版. 北京：科学出版社，2017.

[2] 雷思磊. 自己动手写 CPU[M]. 北京：电子工业出版社，2014.

[3] HARRIS D M. 数字设计和计算机体系结构[M]. 2 版. 陈俊颖，译. 北京：机械工业出版社，2016.

[4] 胡振波. 手把手教你设计 CPU——RISC_V 处理器篇[M]. 北京：人民邮电出版社，2018.

[5] 唐朔飞. 计算机组成原理[M]. 2 版. 北京：高等教育出版社，2008.

[6] 张冬冬，王力生，郭玉臣. 数字逻辑与组成原理实践教程[M]. 北京：清华大学出版社，2018.

[7] 夏宇闻. Verilog 数字系统设计教程[M]. 3 版. 北京：北京航空航天大学出版社，2013.

[8] BHASKER J. Verilog HDL 入门[M]. 3 版. 夏宇闻，甘伟，译. 北京：北京航空航天大学出版社，2008.

[9] 高小鹏. 计算机组成与实现[M]. 北京：高等教育出版社，2019.

[10] 谭志虎，秦磊华，胡迪青. 计算机组成原理实践教程——从逻辑门到 CPU[M]. 北京：清华大学出版社，2018.

[11] RISC-V Foundation. RISC-V Specifications[EB/OL]. [2019-06-25]. https://riscv. org/specifications/.

图 书 资 源 支 持

感谢您一直以来对清华大学出版社图书的支持和爱护。为了配合本书的使用，本书提供配套的资源，有需求的读者请扫描下方的“书圈”微信公众号二维码，在图书专区下载，也可以拨打电话或发送电子邮件咨询。

如果您在使用本书的过程中遇到了什么问题，或者有相关图书出版计划，也请您发邮件告诉我们，以便我们更好地为您服务。

我们的联系方式：

地　　址：北京市海淀区双清路学研大厦 A 座 701

邮　　编：100084

电　　话：010-83470236　010-83470237

资源下载：http://www.tup.com.cn

客服邮箱：tupjsj@vip.163.com

QQ：2301891038（请写明您的单位和姓名）

科技传播・新书资讯



电子电气科技荟

资料下载・样书申请



书圈

用微信扫一扫右边的二维码，即可关注清华大学出版社公众号。